BASIC COMPILER User's Manual

BASIC COMPILER User's Manual

BASIC Compiler Command Format and Switches

Procedures for Using the BASIC Compiler

Sample Compilation

Error Messages

8102-510-01

Microsoft
BASIC Compiler User's Manual

CONTENTS

CHAPTER 1

BASIC COMPILER COMMAND SCANNER

1.1 COMMAND FORMAT

To run the BASIC Compiler, type BASCOM followed by a carriage return. (For users with 32K CP/M systems, type BASCOM32 instead of BASCOM. BASCOM32 is a small loader program which loads BASCOM into the user TPA.) BASIC will return the prompt "*", indicating it is ready to accept commands. To tell the BASIC compiler what to compile and with which options, it is necessary to input a "command string," which is read by the compiler's command scanner. The general format of a BASIC compiler command string is:

```
objprog-dev:filename.ext,list-dev:filename.ext=
   source-dev:filename.ext
```

objprog-dev:
The device on which the object program is to be written.

list-dev:
The device on which the program listing is written.

source-dev:
The device from which the source-program input to BASIC is obtained. If a device name is omitted, it defaults to the currently selected drive.

The available device names with CP/M are:

```
     A:, B:, C:, D:   Disk drives
     HSR:     High speed reader
     LST:     Line printer
     TTY:     Teletype or CRT
```

filename.ext
The filename and filename extension of the object program file, the listing file, and the source file. Filename extensions may be omitted. The default filename extensions with CP/M are:

BAS	BASIC source file
MAC	MACRO-80 source file
REL	Relocatable object file
PRN	Listing file
COM	Absolute file
FOR	FORTRAN-80 source file
COB	COBOL-80 source file

Either the object file or the listing file or both may be omitted. If neither a listing file nor an object file is desired, place only a comma to the left of the equal sign. If the names of the object file and the listing file are omitted, the default is the name of the source file.

Examples:

*=TEST	Compile the program TEST.BAS and place the object in TEST.REL
*,TTY:=TEST	Compile the program TEST.BAS and list program on the terminal. No object is generated.
*TESTOBJ=TEST.BAS	Compile the program TEST.BAS and put object in TESTOBJ.REL
*TEST,TEST=TEST	Compile TEST.BAS, put object in TEST.REL and listing in TEST.PRN
*,=TEST.BAS	Compile TEST.BAS but produce no object or listing file. Useful for checking for errors.

1.1.1 BASIC Compilation Switches

A switch on the end of a compiler command string specifies a special parameter to be used during compilation. Switches are always preceded by a slash (/). More than one switch may be used in the same command. The available switches are:

Switch	Action
/E	The /E switch tells the compiler that the program contains the ON ERROR GOTO statement. If a RESUME statement other than RESUME <line number> is used with the ON ERROR GOTO statement, use /X instead (see below). To handle ON ERROR GOTO properly in a compiled environment, BASIC must generate some extra code for the GOSUB and RETURN statements. Therefore, do not use this switch unless your program contains the ON ERROR GOTO statement. The

/E switch also causes line numbers to be included in the binary file, so runtime error messages will include the number of the line in error.

/X The /X switch tells the BASIC compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements. The /E switch is assumed when the /X switch is specified. To handle RESUME statements properly in a compiled environment, the compiler must relinquish certain optimizations. Therefore, do not use this switch unless your program contains RESUME statements <u>other than</u> RESUME <line number>. The /X switch also causes line numbers to be included in the binary file, so runtime error messages will include the kumber of the line in error.

/N The /N switch prevents listing of the generated code in symbolic notation. If this switch is not set, the source listing produced by the compiler will contain the object code generated by each statement.

/D The /D switch causes debug/checking code to be generated at runtime. This switch must be set if you want to use TRON/TROFF. The BASIC compiler generates somewhat larger and slower code in order to perform the following checks:

1. Arithmetic overflow. All arithmetic operations, integer and floating point, are checked for overflow and underflow.

2. Array bounds. All array references are checked to see if the subscripts are within the bounds specified in the DIM statement.

3. Line numbers are included in the generated binary so that runtime errors can indicate the statement which contains the error.

4. RETURN is checked for a prior GOSUB.

/Z The /Z switch tells the compiler to use Z80 opcodes whenever possible. The generated code is listed using 8080 opcodes except in those cases where Z80 opcodes have been used.

/S The /S switch forces the compiler to write long quoted strings (i.e., more than 4 characters) to the binary file as they are encountered. This allows large programs with many quoted strings to compile in less memory. However, there are two disadvantages:

1. Memory space is wasted if identical, long quoted strings appear in the program.

2. Code generated while the /S switch is set cannot be placed in ROM.

/4 The /4 switch allows the compiler to use the lexical conventions of the Microsoft 4.51 BASIC interpreter. That is, spaces are insignificant, variables with embedded reserved words are illegal, variable names are restricted to two significant characters, etc. This feature is useful if you wish to compile a source program that was coded without spaces, and contains lines such as

```
FORI=ATOBSTEPC
```

Without the /4 switch, the compiler would assign the variable "ATOBSTEPC" to the variable FORI. With the /4 switch, it would recognize it as a FOR statement. It is recommended that such programs be edited to the 5.0 lexical standards, rather than using the /4 switch. Delimiting reserved words with spaces causes no increase in the generated code and greatly improves readability.

/C The /C switch tells the compiler to relax line numbering constraints. When /C is specified, line numbers may be in any order, or they may be eliminated entirely. Lines are compiled normally, but of course cannot be targets for GOTOs, GOSUBs, etc. While /C is set, the underline character causes the remainder of the physical line to be ignored, and the next physical line is considered to be a continuation of the current logical line. NOTE: /C and /4 may not be used together.

Examples:

*,TTY:=MYPRG/N Compile MYPRG.BAS and list the source program on the terminal but without the generated code. Put the object file in MYPRG.REL.

*=TEST/E Compile TEST.BAS. The source file contains an ON ERROR GOTO statement. Put the object file in TEST.REL.

*=BIGGONE/D Compile BIGGONE.BAS and put the object file in BIGGONE.REL. Check for overflow and out-of-bound array subscripts, and include line numbers in the object file.

CHAPTER 2

USING THE BASIC COMPILER

2.1 PROCEDURE

The following steps give the procedure for creating, compiling, and saving BASIC programs using the BASIC compiler and LINK-80 loader on the CP/M operating system.

1. Create a source file
   Create a BASIC source file using the CP/M editor or Microsoft's EDIT-80 Text Editor or Microsoft's BASIC-80 interpreter. Filenames are up to eight characters long, with 3-character extensions. BASIC source filenames should have the extension BAS. (MACRO-80 source filenames should have the extension MAC.)

2. Error check
   Before attempting to compile the program and produce object code for the first time, it is advisable to do a simple syntax check. This will help eliminate the necessity of recompiling later due to syntax errors or other easy-to-fix errors. One way to check for errors is to run the program on Microsoft's BASIC-80 interpreter.

   Another way to perform the error check is to do a compilation without generating an object or listing file. For example, if your BASIC source file is called MAX1.BAS, type the following:

   ```
   A>BASCOM ,=MAX1/N
   ```

   This command compiles the source file MAX1.BAS without producing an object or listing file. (For users with 32K CP/M systems, type BASCOM32 instead of BASCOM. BASCOM32 is a small loader program which loads BASCOM into the user TPA.)

   If necessary, return to the editor (or interpreter) and correct any errors.

3. Compile the source file
   To compile the edited source file and produce an object and listing file, type

```
A>BASCOM MAX1,MAX1=MAX1
```

The compiler will create a REL (relocatable) file called MAX1.REL and a listing file called MAX1.PRN.

4. Load, Execute and Save the Program
To load the program MAX1.REL into memory and execute it, type

```
A>L80 MAX1/G
```

To exit LINK-80 and save a memory image of the object code, type

```
A>L80 MAX1/E
```

When LINK-80 exits, three numbers will be printed: the starting address for execution of the program, the end address of the program and the number of 256-byte pages used. For example

```
[210C 301A 48]
```

Use the CP/M SAVE command to save a memory image. The number of pages used is the argument for SAVE. For example

```
A>SAVE 48 MAX1.COM
```

NOTE

> CP/M always saves memory starting at 100H and jumps to 100H to begin execution. Do not use /P or /D to set the origin of the program or data area to 100H, unless program execution will actually begin at 100H.

The CP/M version of LINK-80 is capable of creating COM files by using the /N switch, (See LINK-80 Switches, Utility Software Manual). In our example,

```
A>L80 MAX1,MAX1/N/E
```

loads and links MAX1.REL, creates the file MAX1.COM for direct execution, and exits to CP/M.

An object code file has now been saved on the disk under the name specified with the LINK-80 /N switch or the CP/M SAVE command (in this case MAX1). To execute the program simply type the program name

```
A>MAX1
```

5. CP/M Command Lines
CP/M command lines and files are supported; i.e., a BASIC, COBOL-80, FORTRAN-80, MACRO-80 or LINK-80 command line may be placed in the same line with the CP/M run command. For example, the command

```
A>BASCOM =TEST
```

causes CP/M to load and run the BASIC compiler, which then compiles the program TEST.BAS and creates the file TEST.REL. This is equivalent to the following series of commands:

```
A>BASCOM
*=TEST
A>
```

2.2 SAMPLE COMPILATION

```
BASCOM Y5.0 - Copyright 1979 (C) by MICROSOFT - 11776 Bytes Free
 0014 0007        00100    '        SAMPLE BASIC COMPILATION
         ** 0014'L00100:
 0014 0007        00200    '
         ** 0014'L00200:
 0014 0007        00300    DEFINT I-N,S
         ** 0014'L00300:
 0014 0007        00400    DIM S(50)
         ** 0014'L00400:
 0014 006D        00500    S(0) = 1 : S(1) = 1
         ** 0014'L00500: LXI      H,0001
         ** 0017'        SHLD     S%
         ** 001A'        SHLD     S%+0002
 001D 006D        00600    FOR I=0 TO 24
         ** 001D'L00600: LXI      H,0000
         ** 0020'        SHLD     I%
         ** 0023'        JMP      I00000
         ** 0026'I00001:
 0026 006F        00700    S(2*(I+1))=S(2*(I+1)-1)+S(2*(I+1)-2)+3
         ** 0026'L00700: LHLD     I%
         ** 0029'        DAD      H
         ** 002A'        DAD      H
         ** 002B'        PUSH     H
         ** 002C'        LXI      D,S%+0002
         ** 002F'        DAD      D
         ** 0030'        MOV      E,M
         ** 0031'        INX      H
         ** 0032'        MOV      D,M
         ** 0033'        XCHG
         ** 0034'        SHLD     T:01
         ** 0037'        POP      H
         ** 0038'        PUSH     H
         ** 0039'        LXI      D,S%
         ** 003C'        DAD      D
         ** 003D'        MOV      E,M
         ** 003E'        INX      H
         ** 003F'        MOV      D,M
         ** 0040'        LHLD     T:01
         ** 0043'        DAD      D
         ** 0044'        INX      H
         ** 0045'        INX      H
         ** 0046'        INX      H
         ** 0047'        SHLD     T:02
         ** 004A'        POP      H
         ** 004B'        LXI      D,S%+0004
         ** 004E'        DAD      D
         ** 004F'        PUSH     H
         ** 0050'        LHLD     T:02
         ** 0053'        XCHG
         ** 0054'        POP      H
         ** 0055'        MOV      M,E
         ** 0056'        INX      H
         ** 0057'        MOV      M,D
```

```
0058 006F        00800    NEXT I
         ** 0058'L00800: LHLD     I%
         ** 005B'        INX      H
         ** 005C'        SHLD     I%
         ** 005F'I00000:
         ** 005F'        LHLD     I%
         ** 0062'        LXI      D,FFE7
         ** 0065'        MOV      A,H
         ** 0066'        RAL
         ** 0067'        JC       I00002
         ** 006A'        DAD      D
         ** 006B'        DAD      H
         ** 006C'I00002: JC       I00001
006F 006F        00900    PRINT "ANSWER =";S(50)
         ** 006F'L00900: CALL     $PR0A
         ** 0072'        LXI      H,<const>
         ** 0075'        CALL     $PV1D
         ** 0078'        LHLD     S%+0064
         ** 007B'        CALL     $PV2C
007E 006F
         ** 007E'        CALL     $END

00000 Fatal Errors
11151 Bytes Free
```

The address in the left-hand column is the current program address. The address in the next column is the current data address.

Note the examples of common subexpression elimination in lines 500 and 700, and constant folding and peephole optimization in line 700.

CHAPTER 3

ERROR MESSAGES

3.1 BASIC COMPILER ERROR MESSAGES

The following errors may occur while a program is compiling. The BASIC compiler outputs the two-character code for the error, along with an arrow. The arrow indicates where in the line the error occurred. In those cases where the compiler has read ahead before it discovered the error, the arrow points a few characters beyond the error, or at the end of the line.

The error codes are as follows:

FATAL ERRORS

Code	Error
SN	Syntax Error. Caused by one of the following: Illegal argument name Illegal assignment target Illegal constant format Illegal debug request Illegal DEFxxx character specification Illegal expression syntax Illegal function argument list Illegal function name Illegal function formal parameter Illegal separator Illegal format for statement number Illegal subroutine syntax Invalid character Missing AS Missing equal sign Missing GOTO or GOSUB Missing comma Missing INPUT Missing line number Missing left parenthesis Missing minus sign Missing operand in expression Missing right parenthesis

```
          Missing semicolon
          Name too long
          Expected GOTO or GOSUB
          String assignment required
          String expression required
          String varible required here
          Illegal syntax
          Variable required here
          Wrong number of arguments
          Formal parameters must be unique
          Single variable only allowed
          Missing TO
          Illegal FOR loop index variable
          Missing THEN
          Missing BASE
          Illegal subroutine name

OM      Out of Memory
          Array too big
          Data memory overflow
          Too many statement numbers
          Program memory overflow

SQ      Sequence Error
          Duplicate statement number
          Statement out of sequence

TM      Type Mismatch
          Data type conflict
          Variables must be of same type

TC      Too Complex
          Expression too complex
          Too many arguments in function call
          Too many dimensions
          Too many variables for LINE INPUT
          Too may variables for INPUT

BS      Bad Subscript
          Illegal dimension value
          Wrong number of subscripts

LL      Line Too Long

UC      Unrecognizable Command
          Statement unrecognizable
          Command not implemented

OV      Math Overflow

/0      Division by Zero

DD      Array Already Dimensioned
```

FN FOR/NEXT Error
  FOR loop index variable already in use
  FOR without NEXT
  NEXT without FOR

FD Function Already Defined

UF Function Not Defined

WE WHILE/WEND Error
  WHILE without WEND
  WEND without WHILE

/E Missing "/E" Switch

/X Missing "/X" Switch

WARNING ERRORS

ND Array Not Dimensioned

SI Statement Ignored
  Statement ignored
  Unimplemented command

3.2 BASIC RUNTIME ERROR MESSAGES

The following errors may occur while a compiled program is executing. The error numbers match those issued by the BASIC-80 interpreter. The compiler runtime system prints long error messages followed by an address, unless /D, /E, or /X is specified. In those cases the error message is followed by the number of the line in which the error occurred.

Number	Message
2	Syntax error A line is encountered that contains an incorrect sequence of characters in a DATA statement.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement
4	Out of data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal function call A parameter that is out of range is passed to a math or string function. An FC error may also occur as the result of: 1. a negative or unreasonably large subscript 2. a negative or zero argument with LOG 3. a negative argument to SQR 4. a negative mantissa with a non-integer exponent 5. a call to a USR function for which the starting address has not yet been given 6. an improper argument to ASC, CHR$, MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO 7. a string concatenation that is longer than 255 characters
6	Floating overflow or integer overflow The result of a calculation is too large to be represented in BASIC-80's number format. If underflow occurs, the result is zero and execution continues without an error.

9 Subscript out of range
An array element is referenced with a subscript that is outside the dimensions of the array.

11 Division by zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power. Machine infinity with the sign of the numerator is supplied as the result of the division, or positive machine infinity is supplied as the result of the involution, and execution continues.

14 Out of string space
String variables exceed the allocated amount of string space.

20 RESUME without error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable error
An error message is not available for the error condition which exists. This is usually caused by an ERROR with an undefined error code.

50 Field overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal error
An internal malfunction has occurred in Disk BASIC-80. Report to Microsoft the conditions under which the message appeared.

52 Bad file number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File not found
A LOAD, KILL or OPEN statement references a file that does not exist on the current disk.

54 Bad file mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file or to execute an OPEN with a file mode other than I, O, or R.

55 File already open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O error
An I/O error occurred on a disk I/O operation. It is a fatal error, i.e., the operating system cannot recover from the error.

58 File already exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk full
All disk storage space is in use.

62 Input past end
An INPUT statement is exeucted after all the data in the file has been INPUT, or for a null (empty) file. To avoid this error, use the EOF function to detect the end of file.

63 Bad record number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or equal to zero.

64 Bad file name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

67 Too many files
An attempt is made to create a new file (using SAVE or OPEN) when all 255 directory entries are full.

Microsoft BASIC Compiler, Version 5.3 July, 1981

ADDENDA TO: The BASIC Compiler User's Manual

There are significant differences between version 5.3 of the Microsoft BASIC Compiler and previous versions. Major differences are listed below:

1. CHAINing with COMMON is supported.

2. Runtime support is now organized so that a single large module contains most of the runtime library.

3. In conjunction with points 1. and 2., now large systems of programs can be created that share common data and use a single runtime environment.

4. Larger programs (16K larger on the average) can be compiled and linked.

5. Programs take up significantly less disk space.

See Appendix D of this manual for a further discussion of these and other changes.

SYSTEM REQUIREMENTS

The Microsoft BASIC Compiler can be used with most microcomputers with a minimum of 48K RAM and one disk drive. We recommend two drives, however, for easier operation. The compiler operates under the CP/M operating system, which is required.

CP/M is a registered trademark of Digital Research

Royalty Information

For those who want to market application programs, use of the BASIC Compiler provides you with three major benefits:

1. Increased speed of execution for most programs,

2. Decreased program size for most programs, and

3. Source code security.

When you distribute a compiled program, you distribute highly optimized machine code, not source code. Consequently, you distribute your program in very compact form and protect your source program from unauthorized alteration.

The policy for distribution of parts of the BASCOM package is as follows:

1. Any application program that you generate by linking to either of the two runtime libraries (BASLIB.REL and OBSLIB.REL) may be distributed without payment of royalties. A copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT, 1981" must be displayed on the media.

2. However, the BRUN.COM runtime module cannot be distributed without first entering into a license agreement with Microsoft for such distribution. A copy of the license agreement can be readily obtained by writing to Microsoft. Also, a copyright notice reading "PORTIONS COPYRIGHTED BY MICROSOFT, 1981" must be displayed on the media.

3. All other software in your BASIC Compiler package cannot be duplicated except for purposes of backing up your software. Other duplication of any of the software in the BASIC Compiler package is illegal.

All of the above information is included in the Non-Disclosure Agreement, which must be signed and returned to Microsoft at the time the BASIC Compiler is purchased. In order to provide you any updates or fixes, we must have your completed form on file. Failure to register and sign the non-disclosure agreement voids any warranty expressed or implied.

CONTENTS

CHAPTER 1

INTRODUCTION

The Microsoft BASIC Compiler is an optimizing compiler designed to complement Microsoft's BASIC-80 interpreter. Since BASIC-80 is the recognized standard for microcomputer BASIC, the BASIC compiler can support programs written for a wide variety of microcomputers.

In addition, the BASIC Compiler allows you to create programs that:

1. Execute faster in most cases than the same interpreted programs,

2. Require less memory in most cases than the same interpreted programs, and are

3. Source-code secure.

These benefits can be critical for real-time applications such as graphics, where execution speed can often make or break an application; business applications, where several CHAINED programs can be supported by a main menu in a single runtime environment; and commercial applications, where software is being sold in a competitive marketplace and source-code security is essential.

There is another major advantage that you gain by owning the compiler. Because the BASIC Compiler has been created to support most of the interpreted BASIC-80 language, the interpreter and the compiler complement each other, and provide you with an extrememly powerful BASIC programming environment. In this environment, you can quickly RUN and debug programs from within BASIC-80, and then later compile those programs to increase their speed of execution and to decrease their space in memory.

Although the language supported by the BASIC Compiler is not identical to that supported by the interpreter; the compiler has been designed so that compatibility is maintained where ever possible. Note also, that the file named BRUN.COM contains the majority of the runtime

environment. For this reason, BRUN.COM is called the runtime module. The runtime module is loaded when program execution begins; later execution of CHAINed programs does not require reloading. This allows you to develop a system of related programs that can all be run using the same runtime environment. The runtime environment required by your program need not be saved on disk as part of your executable .COM file. For a system of four programs, this can save at least 48K of disk space--a substantial savings.

This version (5.3) of the BASIC Compiler is substantially different from previous versions. These differences are summarized in Appendix D.

1.1 HOW TO USE THIS MANUAL

The BASIC Compiler User's Manual is designed for users who are unfamiliar with the compiler as a programming tool. Therefore, this manual provides both a step-by-step introduction and a detailed technical guide to the BASIC Compiler and its use. After a few compilations, the User's Manual then serves as both a refresher on procedures and as a technical reference.

This manual assumes that the user has a working knowledge of the BASIC language. For reference information, consult the BASIC-80 Reference Manual. If you need additional information on BASIC programming, refer to Section 1.5 of this manual, RESOURCES FOR LEARNING BASIC.

Organization

This manual contains the following chapters:

Chapter 1, INTRODUCTION. Provides brief descriptions of the contents of the BASIC Compiler package, and gives a list of references for learning BASIC programming.

Chapter 2, AN INTRODUCTION TO COMPILATION. Gives you an introduction to the vocabulary associated with compilers, a comparison of interpretation and compilation, and an overview of program development with the compiler.

Chapter 3, DEMONSTRATION RUN. Takes you step by step through the compiling, linking, and running of a demonstration program.

Chapter 4, EDITING. Describes how to create a BASIC source program for later compilation, and how to use the %INCLUDE compiler directive.

Chapter 5, DEBUGGING WITH THE INTERPRETER. Describes how to debug the BASIC source file with the BASIC-80 interpreter before compiling it. Note that Chapter 9, A COMPILER/INTERPRETER COMPARISON describes differences between the language supported by the compiler and that supported by the BASIC-80 interpreter.

Chapter 6, COMPILING. Describes use of the BASIC Compiler in detail, including descriptions of the command line syntax and the various compiler options.

Chapter 7, LINKING. Describes how to use LINK-80 to link your programs to needed runtime support. (Note that the Utility Software Manual contains further reference material on LINK-80.)

Chapter 8, RUNNING A PROGRAM. Describes how to run your final executable program.

Chapter 9, A COMPILER/INTERPRETER COMPARISON. Describes all of the language, operational, and other differences between the language supported by the BASIC Compiler and that supported by the BASIC-80 interpreter. It is important to study these differences and to make the necessary editing changes in your BASIC program before you use the compiler.

Chapter 10, ERROR MESSAGES. Describes each error message.

Appendices that show you how to create a system of programs with the BRUN.COM runtime module, and how to generate a ROM-able program are also provided. Two other appendices give you a memory map of the BRUN.COM runtime environment, and describe the differences between this and pre-5.3 versions of the compiler.

NOTATION USED IN THIS MANUAL

For the most part, any punctuation marks or other special characters used, especially in command formats, are to be taken literally. Consider these marks as part of the command format.

However, some special characters used in command formats have special meanings:

capital letters — FOO Indicate that the parameter or command must be entered exactly as shown.

angle brackets — <> Indicate that enclosed text specifies a class of parameters. Any parameter that you enter in this position must be a valid member of that parameter class. Hence, <filename> means that you must enter a legal filename.

Capital letters enclosed by angle brackets are used to specify non-displayable ASCII characters. For example, <CR> specifies entry of a carriage return.

square brackets — [] Indicate that the enclosed parameter is optional. For instance, <filename>[,<filename>] specifies entry of either one filename or two filenames.

ellipses — ... Indicate that the symbols preceding the ellipses can be entered as many times as needed. For example, <filename>... indicates entry of one or more filenames.

1.2 CONTENTS OF THE BASIC COMPILER PACKAGE

The BASIC Compiler Package contains:

One disk containing the following files:

BASCOM.COM - The BASIC Compiler
BRUN.COM - The Runtime Module
BASLIB.REL -The Runtime Library
OBSLIB.REL - The Alternate Runtime Library
BCLOAD - Runtime load information file
L80.COM - The LINK-80 Linking Loader
M80.COM - The MACRO-80 Macro-assembler
CREF.COM - The Cross-reference Utility
LIB80.COM - The Library Manager
DEMO.BAS - A Demonstration program

A Binder with three Manuals including the following:

The BASIC Compiler User's Manual (this manual)
The BASI -80 Reference Manual
The Utility Software Manual

1.3 SOFTWARE

A description follows of the function of the software on your disk:

1. BASCOM.COM - (The BASIC Compiler) Compiles BASIC source files into relocatable and linkable .REL files.

2. BRUN.COM - (The Runtime Module) A single module containing most of the routines called from your compiled .REL file. So that the entire BRUN.COM module is not loaded into memory at linktime, a dummy module that resolves all of the references to routines in BRUN.COM resides in BASLIB.REL.

3. BASLIB.REL - (The Runtime Library) A collection of routines implementing functions of the BASIC language not found in BRUN.COM. Your .REL file may contain calls to these routines.

4. OBSLIB.REL -(The Old Runtime Library) A collection of modules containing routines that are similar to the routines found in BASLIB.REL and BRUN.COM, above. This library should be used for applications that you wish to make ROM-able, or for those that you want to execute as single .COM files without the BRUN.COM runtime module. This library

does not support CHAIN with COMMON, CLEAR, or RUN <linenumber>. Additional differences are described in Chapter 6, Linking.

5. BCLOAD - (Runtime load information file) Tells at what address to load your program at linktime, and where to find BRUN.COM at runtime.

6. L80.COM - (The Linking Loader) Links and loads compiled .REL files, library modules, and assembly language routines to create an executable .COM file.

7. M80.COM - (The Macro-Assembler) Assembles assembly language routines into .REL files that can later be linked to your compiled .REL file.

8. CREF.COM - (The Cross-Reference Utility) Creates a cross-referenced listing of the use of variables in assembly language programs.

9. LIB80.COM - (The Library Manager) Allows you to create and modify user runtime libraries.

10. DEMO.COM - (A Demonstration Program) Used in Chapter 3 to demonstrate program development with the BASIC Compiler.

1.4 DOCUMENTATION

Three manuals come with the BASIC Compiler package: the BASIC Compiler User's Manual (this manual), the BASIC-80 Reference Manual, and the Utility Software Manual. Each manual provides specific information necessary for the successful creation of an executable compiled BASIC program.

THE BASIC COMPILER USER'S MANUAL

This manual is described above in Section 1.1, How To Use This Manual. See that section for more information.

BASIC-80 REFERENCE MANUAL

The BASIC-80 Reference Manual describes syntax and usage of Microsoft's standard BASIC language. This is the language supported by the BASIC Compiler, with the exceptions noted in Chapter 9 of the BASIC Compiler User's Manual. Note that the BASIC-80 interpreter itself is not supplied as part of the BASIC Compiler package.

The BASIC Compiler supports, in some form, all of the statements and commands described in the BASIC-80 manual, except:

AUTO	CLOAD	CSAVE	CONT	DELETE	EDIT
ERASE	LIST	LLIST	LOAD	MERGE	NEW
RENUM	SAVE				

IMPORTANT

> Language, operational, and other differences between the BASIC Compiler and the BASIC-80 interpreter are described in Chapter 9, A BASIC COMPILER/INTERPRETER COMPARISON. You should review the information in that chapter before compiling any of your programs that already run without problem when interpreted by BASIC-80. Only then make any necessary changes.

UTILITY SOFTWARE MANUAL

The Utility Software Manual provides descriptions of the following pieces of software in your BASIC Compiler package:

1. LINK-80

2. MACRO-80

3. LIB-80

4. CREF-80

1.5 RESOURCES FOR LEARNING BASIC

Microsoft provides complete instructions for using the BASIC Compiler. However, no teaching material for BASIC programming has been supplied. The BASIC-80 Reference Manual is strictly a syntax and semantics reference for the Microsoft BASIC-80 language.

CHAPTER 2

INTRODUCTION TO COMPILATION

2.1 COMPILATION VS. INTERPRETATION

A microprocessor can execute only its own machine instructions; it cannot execute BASIC statements directly. Therefore, before a program can be executed, some type of translation must occur from the statements contained in your BASIC program to the machine language of your microprocessor. Compilers and interpreters are two types of programs that perform this translation. This discussion explains the difference between these two translation schemes, and explains why and when you want to use the compiler.

Interpretation

An interpreter performs translation line by line <u>during</u> runtime. To execute a BASIC statement, the interpreter must analyze the statement, check for errors, then perform the BASIC function requested.

If a statement must be executed repeatedly (inside a FOR/NEXT loop, for example), this translation process must be repeated each time the statement is executed.

In addition, BASIC programs are stored as a linked list of numbered lines, and each line is not available as an absolute memory address during interpretation. Therefore, branches such as GOTOs and GOSUBs cause the interpreter to examine every line number in a program, starting with the first, until the line referred to is found.

Similarly, a list of all variables is maintained by the interpreter. When a reference to a variable is made in a BASIC statement, this list must be searched from the beginning until the variable referred to is found. Thus, absolute memory addresses are not associated with the variables in your program.

Compilation

A compiler, on the other hand, takes a source program and translates it into an object file. The object file contains relocatable machine code. All translation takes place before runtime; no translation of your BASIC source file occurs during the execution of your program. In addition, absolute memory addresses are associated with variables and with the targets of GOTOs and GOSUBs, so that lists of variables or of line numbers do not have to be searched during execution of your program.

Note also, that the compiler is an optimizing compiler. Optimizations such as expression re-ordering or sub-expression elimination are made to either increase speed of execution or to decrease the size of your program.

These factors all combine to measurably increase the execution speed of your program. In most cases, execution of BASIC programs is 3 to 10 times faster than execution of the same program under the interpreter. If maximum use of integer variables is made, execution can be up to 30 times faster.

2.2 VOCABULARY

Before you read any farther in this manual, you need to become familiar with some of the vocabulary that is commonly used when discussing compilers.

To begin with, you should understand that a BASIC program is more commonly called a BASIC "source." This source file is the input file to the compiler and must be in ASCII format. The compiler translates this source and creates as output, a new file, called a relocatable "object" file. These two files have the default extensions .BAS and .REL, respectively.

Other terms that you should know are related to stages in the development and execution of a compiled program. These stages are described below:

Compiletime - That period of time during which the compiler is executing, and during which it is compiling a BASIC source file and creating a relocatable object file.

Linktime - That period of time during which the linker is executing, and during which it is loading and linking together relocatable object files and library files.

Runtime - That period of time during which a compiled and linked program is executing. By convention, runtime refers to the execution time of your program and not to the execution time of the compiler or the linker.

You should also learn the following terms pertaining to the linking process and to the runtime support library:

Module - A fundamental unit of code. There are several types of modules, including relocatable and executable modules. Relocatable modules are manipulated by the linker. Your final executable program and BRUN.COM are executable modules. Note that BRUN.COM is special since it is executable only so that you can see its version number. Its main purpose is to serve as a library of routines that can be called at runtime from your compiled program.

Global Reference - A variable name or label in a given module that is referred to by a routine in another module. Global labels are entry points into modules.

Unbound Global Reference - A global reference in a module that is not declared in that module. The linker tries to "resolve" this situation by searching for the declaration of that reference in other modules. These other modules are usually library modules in the runtime library. If the variable or label is found, the address associated with it is substituted for the reference in the first module, and is then said to be "bound." When a variable is not found, it is said to be "undefined."

Relocatable - A module is relocatable if the code within it can be "relocated" and run at different locations in memory. Relocatable modules contain labels and variables represented as offsets relative to the start of the module. These labels and variables are said to be "code relative." When the module is loaded by the linker, an address is associated with the start of the module. The linker then computes an absolute address that is equal to the associated address plus the code relative offset for each label or variable. These new computed values become the absolute addresses that are used in the binary .COM file.

.REL files and library files are all relocatable modules. Note that normally a relocatable module contains global references as well: these are resolved after all local labels and variables have been computed within other relocatable modules. This process of computing absolute relocated values and resolving global references is what is meant by "linking."

Routine - Executable code residing in a module. More than one routine may reside in a module. The BRUN.COM module contains a majority of the library routines needed to implement the BASIC language. A library routine usually corresponds to a feature or sub-feature of the BASIC language.

Runtime Support - The body of routines that may be linked to your compiled .REL file. These routines implement various features of the BASIC language. BRUN.COM, OBSLIB.REL, and BASLIB.REL all contain runtime support routines. See Chapter 6, LINKING, for more information on runtime support.

The BRUN.COM Module - A module containing most of the routines needed to implement the BASIC language. It is a peculiarity of the BRUN.COM module that it is an executable .COM file. BRUN.COM, for the most part, is a library of routines: it is made executable so that you can see the version number of the module.

Use of BRUN.COM gives you the following advantages:

1. True CHAINing is allowed.

2. COMMON can be used to communicate between CHAINed programs, not just between subroutines.

3. Linktime is reduced, since unbound globals do not have to be searched for in multiple library modules.

4. The BRUN.COM module is not explicitly loaded at link-time, allowing considerably larger programs to be linked and loaded, since an extra 16K is not contained in your final .COM file.

Note, however, that BRUN.COM must be accessible on disk when you execute your final .COM file.

The BASLIB.REL Runtime Library- A collection of modules containing routines for BASIC functions that often are not used in a program. The transcendental functions, the PRINT USING function, some error handling code, and other miscellaneous functions are contained in this library. These functions are linked to your program only if needed.

BASLIB.REL also contains a module consisting of all the global references in the BRUN.COM module. This module exists so that the routines in BRUN.COM can be linked to your compiled .REL file without BRUN.COM itself being brought into memory at linktime.

The OBSLIB.REL Runtime Library - A collection of modules containing routines almost identical in function to similar routines contained in BRUN.COM and BASLIB.REL. However,

this library does not support the CLEAR command, the RUN <line-number> option of the RUN command, and COMMON between CHAINed subprograms. It does support a version of CHAIN that is semantically equivalent to the simple RUN command.

Link Loading - The process in which the LINK-80 linking loader loads modules into memory, computes absolute addresses for labels and variables in relocatable modules, and then resolves all global references by searching the BASLIB.REL runtime library. After loading and linking, the linker saves the modules that it has loaded into memory as a single .COM file on your disk. This entire process is called link loading.

Complete understanding of all the above terms is not essential for continued reading. You may want to refer back to these terms later, as you become familiar with the compiler and with the linker. We now discuss the program development process.

2.3 THE PROGRAM DEVELOPMENT PROCESS

This discussion of the program development process is keyed to figure 2.1. Use it for reference when reading this text.

Program development begins with (1.) the creation of a BASIC source file. The best way to create a BASIC source file is with the editing facilities of BASIC-80, although you can use any general purpose text editor if you wish. Note that files must be SAVEd with the ,A option from BASIC-80.

Once you have written a program, you should use BASIC-80 (2.) to debug the program by RUNning it to check for syntax and program logic errors. There are a few differences in the languages understood by the compiler and the interpreter, but for the most part they are identical. Because of this similarity, running a program provides you with a much quicker syntactic and semantic check of your program than does compiling, linking, and finally executing a program. Therefore, you should strive to make the interpreter your chief debugging tool.

After you have debugged your program with the interpreter, (3.) compile it to check out differences that may exist between interpreted and compiled BASIC. The compiler flags all syntax errors as it reads your source file. If compilation is successful, the compiler creates a relocatable .REL file.

The .REL file is not executable, and needs to be linked to the BASLIB.REL runtime library. You may want to include your own assembly language routines to increase the speed of execution of a particular algorithm, or to handle operations that require a more intimate relationship with the microprocessor. For these cases, use MACRO-80, the macro-assembler, (4.) to assemble routines that you can later link to your program. Similarly, separately compiled Microsoft FORTRAN subroutines can be linked to your program. (FORTRAN is a separate product available from Microsoft. Macro-80 is discussed in the Utility Software Manual.) The linker (5.) links all modules needed by your program, and produces as output an executable object file with .COM as the default extension. This file can be (6.) executed like any .COM file by simply typing the file's base name (the file name less its .COM extension).

This program development process is demonstrated in the following chapter, Chapter 3, DEMONSTRATION RUN.

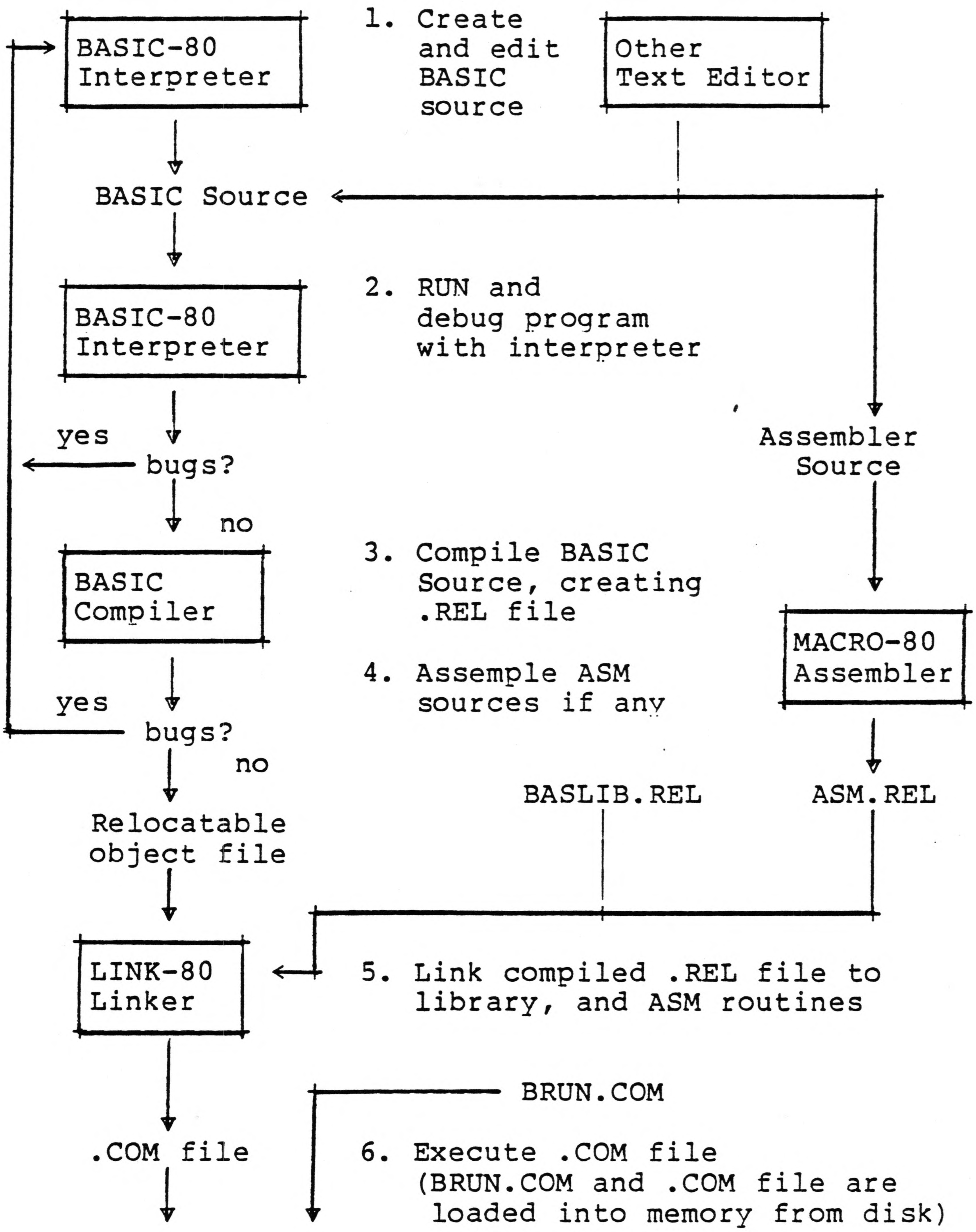


Figure 2.1 The Program Development Process

CHAPTER 3

DEMONSTRATION RUN

IMPORTANT

Before beginning this demonstration run, make a backup copy of your BASIC Compiler disk. Next, COPY CP/M on to your copied disk so that it can be booted up by itself. Store your master disk in a safe place and work with this backup copy.

This chapter provides step by step instructions for using the BASIC Compiler. These steps are outlined using a demonstration program.

We strongly recommend that you compile the demonstration program <u>before</u> compiling any other programs, because this demonstration run gives you an overview of the compilation process. Also, you should read Chapters 4 through 9. They contain important information that is crucial to successful development of a program.

If you enter commands exactly as described in this chapter, you should have a successful session with the BASIC Compiler. If a problem does arise, check and redo each step carefully.

The five steps in developing a program with the BASIC Compiler are:

1. Editing (entering and correcting the BASIC program)

2. Debugging with the Interpreter (using BASIC-80 to RUN your program)

3. Compiling (creating a relocatable object file)

4. Linking (creating an executable object file)

5. Running (executing the program)

Because we have prepared a special debugged demonstration program on disk, you do not have to perform the first two steps in the program development process. Therefore, the demonstration run begins with compilation. Note that we have SAVEd the demonstration program on disk with the ,A option, since all files must be in ASCII format to be readable by the compiler.

3.1 COMPILING

To begin compiling a program, insert a copy of your BASCOM disk in drive A: and boot up your system. The BASCOM disk contains all of the files that you need to carry out this demonstration run, including the demonstration program named DEMO.COM. In this demonstration all files produced by the compiler and by the linker will be placed on this disk. Perform the following steps to compile your program:

1. <u>Enter the BASIC Compiler command line</u>.

   Invoke the compiler by typing:

   ```
   BASCOM DEMO,DEMO=DEMO
   ```

   This command line begins compilation of the source file. The source file is the last parameter on the command line, and the .BAS default extension is assumed.

   The compiler generates relocatable object code that is stored in the file specified by the first parameter on the command line. This file is created with the default .REL extension.

   At the same time, a listing file is written out to your disk. Its file name is that specified by the second parameter on the command line (following the comma). This file is created with the default .PRN extension.

2. Look for error messages.

   When the compiler has finished, it displays the message "00000 FATAL ERROR(S)", and program control is returned to CP/M.

   At this point, you should see two new files listed in the A: directory: DEMO.REL and DEMO.PRN.

3. Delete the listing file.

   You may want to view or print out the listing file (DEMO.PRN) at this juncture in the demonstration run. In any event, you should delete the listing file to gain additional disk space. To do this, type:

   ```
   ERA DEMO.PRN
   ```

Further information on listing files is given in Chapter 6, COMPILING. You are now ready for the next step--Linking.

3.2 LINKING

Linking is accomplished with the LINK-80 linking loader (the file named L80.COM). Perform the following steps to link DEMO.REL to needed runtime support.

1. Invoke LINK-80.

   To invoke LINK-80, simply type:

   ```
   L80
   ```

   Your computer will search your disk for LINK-80, load it, and then return the asterisk (*) prompt.

   If you want to stop the linking process, and you have entered only L80 and nothing more, you can exit to CP/M by entering Control-C.

2. Enter the filename(s) you want loaded and linked.

   LINK-80 performs the following operations:

   Loads relocatable object (.REL) modules,

   Computes absolute addresses for all local references within modules,

   Resolves all unbound global references between loaded modules, and

   Saves the linked and loaded modules as an executable (.COM) file on disk.

   After the asterisk prompt, type the following line to cause loading, linking, and saving of the program DEMO.COM:

   ```
   DEMO,DEMO/N/E
   ```

   The first part of the command (DEMO) causes loading of the program called DEMO.REL. The /N switch causes an executable image of the linked file to be saved on your disk with the Name DEMO.COM. This occurs after an automatic search of the BASLIB.REL runtime library. The file is only saved after a /E or a /G switch is entered on the command line. You may enter as many command lines as needed before you enter a /E or /G switch. Note that the /E switch, causes an Exit back to CP/M. If you substitute /G for /E here, you cause execution of the new .COM file after linking. In either case, BASLIB.REL is automatically searched to satisfy any unbound global references before linking ends.

3. Wait.

   The linking process requires several minutes. During this time, the following messages will appear on your screen:

   ```
   DATA  <program-start>  <program-end>  <bytes>

   <free-bytes> BYTES FREE
   [<start-address> <program-end> <num-of-pages>]
   ```

   This information is described in Chapter 7, LINKING.

4. Examine your directory

   Type as follows:

   ```
   DIR A:
   ```

   You should see a file named DEMO.COM. This is an executable file.

3.3 RUNNING A PROGRAM

Once you have compiled and linked your program, it is simple to run it. From CP/M, enter the program filename, less its .COM extension. In the case of DEMO.COM, type:

```
DEMO
```

The speed of execution of your program should be quite fast relative to execution of the same program with the BASIC-80 interpreter. Compare speeds of execution by running the BASIC source program with the interpreter.

LEARNING MORE ABOUT DEVELOPING A PROGRAM

You have successfully developed and run a simple BASIC program. You are now ready to learn the more technical details that you need to know to compile other BASIC programs. Chapters 4-8 contain more extensive descriptions of each of the steps you followed in this chapter. Chapter 9 describes all of the language, operational, and other differences between the BASIC Compiler and the BASIC interpreter.

CHAPTER 4

EDITING A SOURCE PROGRAM

The creation of your BASIC source program requires the use of a text editor. Most any text editor will do, but the obvious choice is the line editor available from within BASIC-80. If you have previous experience with BASIC-80, then there is little need to learn how to use a new editor.

It is important to note that the compiler expects its source file in ASCII format. If you edit a file from within BASIC-80, it must be SAVEd with the ,A option; otherwise, the compiler will attempt to read a tokenized encoding of your BASIC program. For more information on editing, saving, and loading files with BASIC-80, you should refer to the BASIC-80 Reference Manual.

The BASIC Compiler supports a useful feature that is not available when you run a BASIC program under the interpreter. This is the %INCLUDE <filename> compiler directive. It is called a compiler directive rather than a BASIC command because it is not really a part of the BASIC language. Rather, it is a command to the compiler, thus its distinctive "%" prefix.

The %INCLUDE <filename> directive allows you to switch compilation of BASIC source files in mid-stream. It switches from the source file you specify on invoking the compiler, to the file you specify as <filename> in the %INCLUDE directive (the <filename> parameter does <u>not</u> require surrounding quotes). When compilation of the external file is complete, the compiler switches back to the original BASIC source and continues compilation.

This process is equivalent to having the text of <filename> expanded at the location of the %INCLUDE directive in your BASIC source. (Note that %INCLUDEs cannot be nested.) Any file that is %INCLUDEd in your BASIC program is called an INCLUDE file. All INCLUDE files must be SAVEd with the ,A option if edited within BASIC-80. If you use another editor, this is not the case.

You may want to create %INCLUDE files for any COMMON declarations existing in more than one program, or for subroutines that you might have in an external library of subroutines. Note that the BASIC-80 interpreter does not support the %INCLUDE directive, thus a syntax error occurs when %INCLUDE is encountered during interpretation.

To make INCLUDE files easily included in large numbers of programs, you may want to edit the INCLUDE file so that it has no line numbers. The compiler supports sequences of lines without line numbers if the /C is used during compilation. However, the BASIC-80 interpreter does not allow you to create lines without line numbers, so you need an external editor to do so. Also, line numbers must exist for any lines that are targets for GOTOs or GOSUBs.

A word here about the differences between the languages supported by the interpreter and the compiler. The interpreter supports a number of editing and file manipulation commands that are useful mainly when creating a program. Examples are LOAD, SAVE, LIST, and EDIT. These are operational commands not supported by the compiler. Some differences also exist for some of the other statements and functions. Realize that it is during editing that you should account for language differences. See Chapter 9, A COMPILER/INTERPRETER COMPARISON for a full description of these differences.

Note also, that the interpreter cannot accept <u>physical</u> lines greater than 254 characters in length. A physical line is the unit of input to the interpreter. Interpreter logical lines can contain as many physical lines as desired.

In contrast to the interpreter, the BASIC Compiler accepts <u>logical</u> lines of up to only 253 characters in length. If you are using an external editor, you can create logical lines containing sequences of physical lines by ending your lines with an underscore. The underscore removes the significance of the carriage return in the <CR><LF> sequence that ends each line (underscore characters in quoted strings do not count). This results in just a linefeed being presented to the compiler. The linefeed, <LF>, is the line continuation character understood by the compiler and the interpreter. The ASCII key code for a linefeed is Control-J.

CHAPTER 5

DEBUGGING WITH THE BASIC INTERPRETER

You should use BASIC-80 to interpret your BASIC source, and thus to check for syntax and program logic errors. Note that debugging with BASIC-80 is an optional step.

It is possible that you do not have the Microsoft BASIC-80 interpreter, and only own the BASIC Compiler. If this is the case, you must edit your program with any general purpose text editor and check for any errors at compiletime. We strongly urge you to complement the compiler with the BASIC-80 interpreter because the combination of the two gives you an extremely powerful BASIC programming environment.

You may use some commands or functions in your compiled program that execute differently with the interpreter. In those cases, you need to use the compiler for debugging. Note that %INCLUDE is the only statement supported by the compiler that is not supported in some form by the BASIC-80 interpreter. Also, the interpreter does not support double precision transcendental functions as does the BASIC Compiler.

Nevertheless, the language supported by the compiler is intended to be as similar to BASIC-80 as possible. This allows you to make BASIC-80 your prime debugging tool, and to save you debugging time by avoiding lengthy compilations and links. Also, the RUN, CONT, and TRON/TROFF statements make BASIC-80 a very powerful interactive debugging tool. See your BASIC-80 Reference Manual for more information on these statements.

Note that the interpreter stops execution of a program when an error is encountered. Any subsequent errors are not caught until the first detected error is corrected and the program re-RUN. This differs from the compiler where all lines are scanned and all detected errors are reported at compiletime.

CHAPTER 6

COMPILING

After creating a BASIC source program that you have debugged with the interpreter, your next step is compilation. This chapter covers:

1. Compiler command line syntax,
2. Sample compiler invocations, and
3. Compiler switches.

6.1 COMMAND LINE SYNTAX

Unlike the BASIC-80 interpreter, the compiler is not interactive. It accepts only a single command line containing filenames and extensions, appropriate punctuation, optional device designations, and switches. The placement of these elements when entering the command line determines which processes the compiler performs. To allow users of single-drive system configurations to use the compiler, the command line can be separated into two command lines if desired: one to invoke the compiler and the other to specify compilation parameters.

The general format for the BASIC Compiler command line is:

```
[<objectfile>][,[<listfile>]]=<sourcefile>
|____________________________| |__________|
              |                      |
         output files           input file
```

<objectfile> Specifies the name of the relocatable (.REL) object file,

<listfile> Specifies the name of the listing (.PRN) file,

<sourcefile> Specifies the name of the BASIC (.BAS) source file.

When filenames are entered as parameters, the compiler reads them according to the syntax described above, and assigns them to the appropriate input and output parameters.

Note that the above syntax is concise and accurate, but can be fairly cryptic. We will clear up questions in the following paragraphs, by examining several sample compiler invocations.

6.2 SAMPLE COMPILER INVOCATIONS

You can specify on the compiler command line, creation of four possible combinations of files. These are listed below:

1. A .REL (relocatable object) file only.

2. A .PRN (listing) file only.

3. Both a .REL and a .PRN file.

4. Neither a .REL file nor a .PRN file.

Sample compiler invocations are given below for these combinations of file productions.

1. <u>How to Generate both Object and Listing Files</u>

   To generate <u>both</u> object and listing files, invoke the compiler as shown below:

   ```
   BASCOM  <objectfile>,<listfile>=<sourcefile>
   ```

   The <objectfile> and <listfile> parameters default to the currently logged drive. You may prefix the file specifications for these parameters with optional device designations.

   At the end of your compilation, the following message is displayed:

   ```
   <number-of-errors> FATAL ERROR(S)
   <free-bytes>       BYTES FREE
   ```

2. <u>How to Generate an Object (.REL) File Only</u>

   The simplest way to create only a .REL file is to invoke the compiler as shown below:

   ```
   BASCOM  =<sourcefile>
   ```

   The above example creates an <objectfile> (not explicitly specified) on the same disk as that containing the <sourcefile>. The <objectfile> will have the same base name as your <sourcefile>. For example, if your <sourcefile> is named A:PROG.BAS, then the <objectfile> will be created with the name A:PROG.REL. Another way to generate only an <objectfile> is to enter:

   ```
   BASCOM  <objectfile>=<sourcefile>
   ```

   In this invocation, <objectfile> defaults to the disk in the currently logged drive. This may or may not be the disk on which <sourcefile> resides. An optional device designation may also be given to either <objectfile> or <sourcefile>.

3. How to Generate a Listing (.PRN) File Only

To create only a listing file, invoke the BASIC Compiler as follows:

```
BASCOM ,<listfile>=<sourcefile>
```

The generated <listfile> contains a line-by-line listing of the BASIC source. Also, the object code generated for each BASIC statement is disassembled and listed along with the corresponding BASIC statements in your program. If you use the /N compiler switch described later in this section, listing of the object code is suppressed. Note that the actual .REL file is not in a human-readable form.

As an alternative, you may have the listing file printed out on a line printer. There are two ways to do this. The first way is to enter Control-P (to turn on the printer), then enter TYPE DEMO.PRN. The listing file is simultaneously printed on the line printer and displayed on your screen. When the file has been printed, enter Control-P again (to turn off the printer).

Another way to print out a listing file is to enter the command line once again, but this time with the name of the line printer device (LST:) in place of the listing filename:

```
BASCOM ,LST:=<sourcefile>
```

The second method is the faster of the two since it does not require the creation of a disk file.

When you examine your listing, notice the two hexadecimal numbers preceding each line of the source program. The first number is the relative address of the code associated with that line, using the start of the program as 0. The second number is the cumulative data area needed so far during the compilation. These two columns are totaled at the end of the listing. The left column total is the actual size of the generated .REL file in bytes. The right column total is the total data area required in bytes.

4. How to Suppress Generation of Any Output Files

To perform a syntax check of your <sourcefile>, and to suppress generation of either an <objectfile> or a <listfile>, invoke the compiler as follows:

```
BASCOM  ,=<sourcefile>
```

In the above example, the compiler simply compiles the source program and reports the number of errors and the number of free bytes. This is the fastest way to perform a syntax check of your program with the compiler. RUNning a program with the interpreter allows you to perform an accurate syntax check only insofar as the language of the BASIC-80 supports the same language as the BASIC Compiler.

You may want to create output files on a disk other than the defaults provided by the compiler, or you may want to create output files with different extensions or base names than that of of your BASIC source file. To do so, you must actually specify the filenames with the desired extensions or device designations, as described below:

Filename Extensions

You may append up to three-characters to filenames as filename extensions. These extensions may contain any alphanumeric character, given in any position in the extension. Lowercase letters are converted to uppercase. Extensions must be preceded by a period (.).

Keep in mind that the BASIC Compiler and L80 recognize certain extensions by default. If you give your filenames unique extensions, you must always remember to include the extension as part of the filename for any filename parameter.

When filename extensions are omitted, default extensions are assumed.

The relevant default filename extensions under CP/M are:

EXTENSION	TYPE OF FILE
.BAS	BASIC source file
.REL	Relocatable object file
.COM	Executable object file
.PRN	Listing file
.MAC	MACRO-80 source file

Device Designations

Each command line field may include device designations that instruct the compiler where to find files or where to output files.

The device designation is placed in front of a filename. For example:

```
B:DEMO
```

A device designation may be up to three alphanumeric characters. Note also that the device name must always include the colon (:).

For the input file, (the <sourcefile>), the device designation indicates from which device the file is read. For output files (<objectfile>, <listfile>), the device designation indicates where the files are written.

Device names supported under CP/M are:

DESIGNATIONS	DEVICES
A:, B:, C:, etc.	Disk Drives
LST:	Line Printer
TTY:	CRT (or Teletype)

When device names are omitted, the command scanner defaults to the currently logged disk drive. The only exception to this occurs if a drive is specified as the device for <sourcefile>, but no filenames are specified for <objectfile> or <listfile>. In this case, the compiler writes the output files to the drive specified for the <sourcefile>.

Take for example, the following command line:

```
BASCOM =B:DEMO
```

This command line directs the compiler to write the object file to the disk in drive B:, regardless of the location of the currently logged drive.

In all other cases, the default device is the currently logged drive. This may, or may not be the disk on which the compiler resides.

For instance, in the following examples, if A: is the currently logged drive, then the output files are written to drive A:.

```
BASCOM DEMO,DEMO=B:DEMO
BASCOM ,DEMO=B:DEMO
```

When the compiler has finished compilation, it exits to C/PM and the currently logged drive.

Device Names as Filenames

Giving a device name in place of a filename is a command line option. The result of this option depends on which device you specify, and for which command line parameter. Figure 6.1 illustrates the possibilities:

DEVICE	<objectfile>	<listfile>	<sourcefile>
A:, B:, C:, D:	writes file to drive specified	writes file to drive specified	N/A
LST:	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY:	N/A (unreadable file format)	"writes" listing to CRT	Reads statements from keyboard

N/A = Not Allowed

Figure 6.1 Effects of Using Device Designations in Place of File Names

Of special interest is the interactive ability you gain by using the ,TTY:=TTY: command line. In this mode, you can type single BASIC statements at your terminal to check them individually for syntax errors. No disk files are created or read.

6.3 COMPILER SWITCHES

In addition to specifying filenames, extensions, and devices to direct the compiler to produce object and listing files, you can direct BASCOM to perform additional or alternate functions by adding switches to the command line.

Switches may be placed after source file names or after other switches, as in the following command line:

```
BASCOM FOO,FOO=FOO/T/4/X
```

Switches signal special instructions to be used during compilation. The switch tells the compiler to "switch on" a special function or to alter a normal compiler function. More than one switch may be used, but all must begin with a slash (/). Do not confuse these switches with the linker switches.

Compiler switches fall into one of three categories:

1. Conventions

2. Error Trapping

3. Special Code

Conventions

The BASIC Compiler allows you to specify which of two lexical and execution conventions you want applied during compilation: version 4.51 or version 5.0. You need to use the lexical convention switches only if you have older programs that you are trying to convert to version 5.0 BASIC conventions. You specify which conventions you want with either or both of the switches /4 and /T.

Error Trapping

If your BASIC source program contains error trapping routines that involve the ON ERROR GOTO statement plus some form of a RESUME statement, you need to use one of the two error trapping switches, /E and /X. Error trapping routines require line numbers in the binary (.REL) file. If you do not use one of the error trapping switches, the compiler does not place line numbers in the binary file, and a fatal compiler error will result.

Special Code

The BASIC Compiler can generate special code for special uses or situations. Be aware that some of these special code switches cause BASIC Compiler to generate larger and slower code. Examples of special code switches are /D, /S, and /O.

Let's go over the compiler switches by category. First, we'll give you a chart that summarizes the function of each switch. Following that, you'll find detailed descriptions of each switch.

Table 6.1 Compiler Switches

CATEGORY	SWITCH	ACTION
Conventions	/4	Use Microsoft 4.51 lexical conventions (not allowed together with /C).
	/T	Use 4.51 execution conventions.
	/C	Relax line numbering constraints (Not allowed together with /4).
		*Use /4/T together for 4.51 lexical and execution conventions.
Error Trapping	/E	Program has ON ERROR GOTO with RESUME <line number>.
	/X	Program has ON ERROR GOTO with RESUME, RESUME 0, or RESUME NEXT.
Special Code	/Z	Use Z80 opcodes.
	/N	Suppress listing of disassembled object code in the listing file.
	/O	Substitute the OBSLIB.REL runtime library for BASLIB.REL as the default runtime library searched by the linker after a linker /E or /G switch.
	/D	Generate debug code for runtime error checking.
	/S	Write quoted strings to .REL file on disk and not to data area in RAM.

Each of the switches shown in table 6.1 is explained in detail in the following pages.

CONVENTIONS

The convention switches may be given together (/4/T) to request 4.51 lexical and execution conventions. The individual action of each switch is described below:

Switch	Action
/4	The /4 switch directs the compiler to use the lexical conventions of the Microsoft 4.51 BASIC-80 interpreter. Lexical conventions are the rules that the compiler uses to recognize the BASIC language.

The following conventions are observed:

1. Spaces are not significant.
2. Variables with embedded reserved words are illegal.
3. Variable names are restricted to two significant characters.

The /4 switch is needed to correctly compile a source program in which spaces do not delimit reserved words, as in the following statement.

```
FORI=ATOBSTEPC
```

Without the /4 switch, the compiler would assign the variable "ATOBSTEPC" to the variable "FORI". With the /4 switch set, the compiler recognizes the line as a FOR statement.

We recommend that you edit such programs to 5.0 lexical standards, rather than compile them with the /4 switch. Delimiting reserved words with spaces causes no increase in generated code while greatly improving program readability.

NOTE

The /4 and /C switches may not be used together.

/T The /T switch tells the compiler to use BASIC-80 Version 4.51 execution conventions. Execution conventions refer to the implementation of BASIC functions and commands and what they actually do at runtime. With /T specified, the following 4.51 execution conventions are switched on:

1. FOR/NEXT loops are always executed at least one time.

2. TAB, SPC, POS, and LPOS perform according to 4.51 conventions.

3. Automatic floating point to integer conversions use truncation instead of rounding, except in the case where a floating point number is being converted to an integer in an INPUT statement.

4. The INPUT statement leaves the variables in the input list unchanged if only a carriage return is entered. If a "?Redo from start" message is issued, then a valid input list must be given. A carriage return in this case generates another "?Redo from start" message.

/C The /C switch tells the compiler to relax line numbering constraints. When /C is specified, line numbers in your source file may be in any order, or they may be eliminated entirely.

With /C, lines are compiled normally, but unnumbered lines cannot be targets for GOTOs or GOSUBs. Be aware that while /C is set, the underline character causes the remainder of the physical line to be ignored. Also, /C causes the underline character to act as a line feed so that the next physical line becomes a continuation of the current logical line. (See Chapter 4 for more information on physical and logical lines.)

There are three advantages to using /C:

1. Elimination of line numbers increases program readability.

2. The BASIC Compiler optimizes over entire blocks of code rather than single lines (for example in FOR...NEXT loops.)

3. BASIC source code can more easily be included in a file with %INCLUDE.

Note that /C and /4 may not be used together.

ERROR TRAPPING

The error trapping switches allow you to use ON ERROR GOTO statements in your program. These statements can aid you greatly in debugging your BASIC programs. Note, however, that extra code is generated by the compiler to handle ON ERROR GOTO statements.

Switch	Action
/E	The /E switch tells the compiler that the program contains an ON ERROR GOTO/RESUME <line-number> construction. To handle ON ERROR GOTO properly, the compiler must generate extra code for the GOSUB and RETURN statements. Also a line number address table (one entry per line number) must be included in the binary file, so that each runtime error message includes the number of the line in which the error occurs. To save memory space and execution time, do not use this switch unless your program contains an ON ERROR GOTO statement.

NOTE

If a RESUME statement other than RESUME <line-number> is used with the ON ERROR GOTO statement, use the /X switch instead.

Switch	Action
/X	The /X switch tells the BASIC Compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements.

The /X switch performs all the functions of the /E switch, so the two need never be used at the same time. For instance, the /X switch, like the /E switch, causes a line number address table (one entry per statement) to be included in your binary object file, so that each runtime error message includes the number of the line in which the error occurs. Nevertheless, the /X switch performs additional functions not performed by the /E switch.

Note that to handle RESUME statements properly, the compiler cannot optimize across statements. Therefore, do not use /X unless your program contains RESUME statements other than RESUME <line-number>.

SPECIAL CODE

Switch	Action
/Z	The /Z switch tells the compiler to use Z80 opcodes whenever possible. When the /Z switch is set, additional Z80 opcodes are allowed, and Z80 mnemonics are used when listing these instructions. All other opcodes are listed using 8080 mnemonics.
/N	The /N switch suppresses listing of the disassembled object code for each source line. Instead, you get a simple BASIC source listing plus the relative locations of your code and the size of your accumulated data area. If this switch is not set, the source listing produced by the compiler contains the disassembled object code generated by each statement. Use this switch when you want a shorter listing file, and want to list your BASIC source file along with the code relative locations of your program and the size of your accumulated data area.
/O	The /O switch tells the compiler to substitute the OBSLIB.REL runtime library for BASLIB.REL as the default runtime library searched by the linker. When you use this switch you cannot use the BRUN.COM module. Note that you can create ROM-able code when you link to OBSLIB.REL, something you cannot do if you link to BASLIB.REL. Also, .COM files created by linking to OBSLIB.REL do not need BRUN.COM on disk at runtime.
/D	The /D switch causes debugging and error handling code to be generated at runtime. Use of /D allows you to use TRON and TROFF in the compiled file. Without /D set, TRON and TROFF are ignored.

With /D, the BASIC Compiler generates somewhat larger and slower code to perform the following checks:

1. Arithmetic overflow. All arithmetic operations, integer and floating point, are checked for overflow and underflow.

2. Array bounds. All array references are checked to see if the subscripts are within the bounds specified in the DIM statement.

3. Line numbers. The generated binary code includes line numbers so that the runtime error listing can indicate on which line each error occurs.

4. RETURN. Each RETURN statement is checked for a prior GOSUB statement.

Without the /D switch set, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not generate error messages at compile time. At runtime, no error messages are generated either, and erroneous program execution results. Use the /D switch to make sure that you have thoroughly debugged your program.

/S The /S switch forces the compiler to write quoted strings greater than 4 characters in length to your .REL file on disk as they are encountered, rather than retaining them in memory during the compilation of your program. If this switch is not set, and your program contains a large number of long quoted strings, you may run out of memory at compiletime.

Although the /S switch allows programs with many quoted strings to take up less memory at compiletime, it may increase the amount of memory needed in the runtime environment, since multiple instances of identical strings will exist in your program. Without /S, references to multiple identical strings are combined so that only one instance of the string is necessary in your final compiled program.

CHAPTER 7

LINKING

To load and link a compiled program, use the Microsoft LINK-80 Linking Loader. Refer to the LINK-80 section of the Utility Software Manual for information on how to use the linker, before you read this chapter. This chapter supplements the Utility Software Manual, by providing:

1. Sample linker sessions,

2. A discussion of linking compiled BASIC programs, and

3. A discussion of the BASIC runtime support environment.

We begin with some sample linker sessions.

7.1 SAMPLE LINKER SESSIONS

A simple link might look like this on your screen:

```
>L80
*PROG.COM/N,PROG.REL/E
```

The caret (>) is the CP/M prompt; the asterisk (*) is the linker prompt. Note that linker switches have no relation whatsoever to the compiler switches discussed in the preceding chapter.

If you use default extensions, a link session might look like this:

```
>L80
*PROG/N,PROG/E
```

The L80 invocation line can also be used for specifying linker parameters. So, the following command would perform the same functions as the preceding example:

```
>L80 PROG/N,PROG/E
```

In any of the above cases, the /E switch tells the linker to exit to CP/M and store a .COM file on disk. Before exiting, the linker automatically searches BASLIB.REL on the currently logged drive for any as yet undefined global references. The final linked .COM file has the name specified by your last <filename>/N command. The /N switch is essential if you want to create a .COM file.

The /G switch is similar to the /E switch. The only difference between the two is that the /G switch causes execution of the .COM file after it is stored on disk. In either case, you must specify the name of the file to store on disk. If you do not, no .COM file is stored.

If you choose to link an assembly language routine to your BASIC program, a sample linker invocation might look like this:

```
>L80
*PROG,MYASM,PROG/N/E
```

In the above case, MYASM.REL is the name of the assembly language routine and PROG.REL is the name of your program. The routine MYASM.REL cannot be assembled with an END <label> statement since the linker will assume that <label> is the start address of a separate program. The linker will refuse to link two programs together since their two separate start addresses will conflict.

When you link a BASIC .REL file to BASLIB.REL, the BCLOAD file must be on disk in the currently logged drive. If it is not, the following error message is generated:

```
?BCLOAD not found, please create header file
```

More information about BCLOAD can be found later in this chapter.

When your linking session is complete, the following message is generated:

```
DATA  <program-start>  <program-end>  <bytes>

<free-bytes> BYTES FREE
[<start-address> <program-end> <num-of-pages>]
```

The values displayed provide the information shown in Figure 7.1 for a program linked to BASLIB.REL and using BRUN.COM. If you link to OBSLIB.REL and use the /P and /D linker switches, some of this information is not accurate.

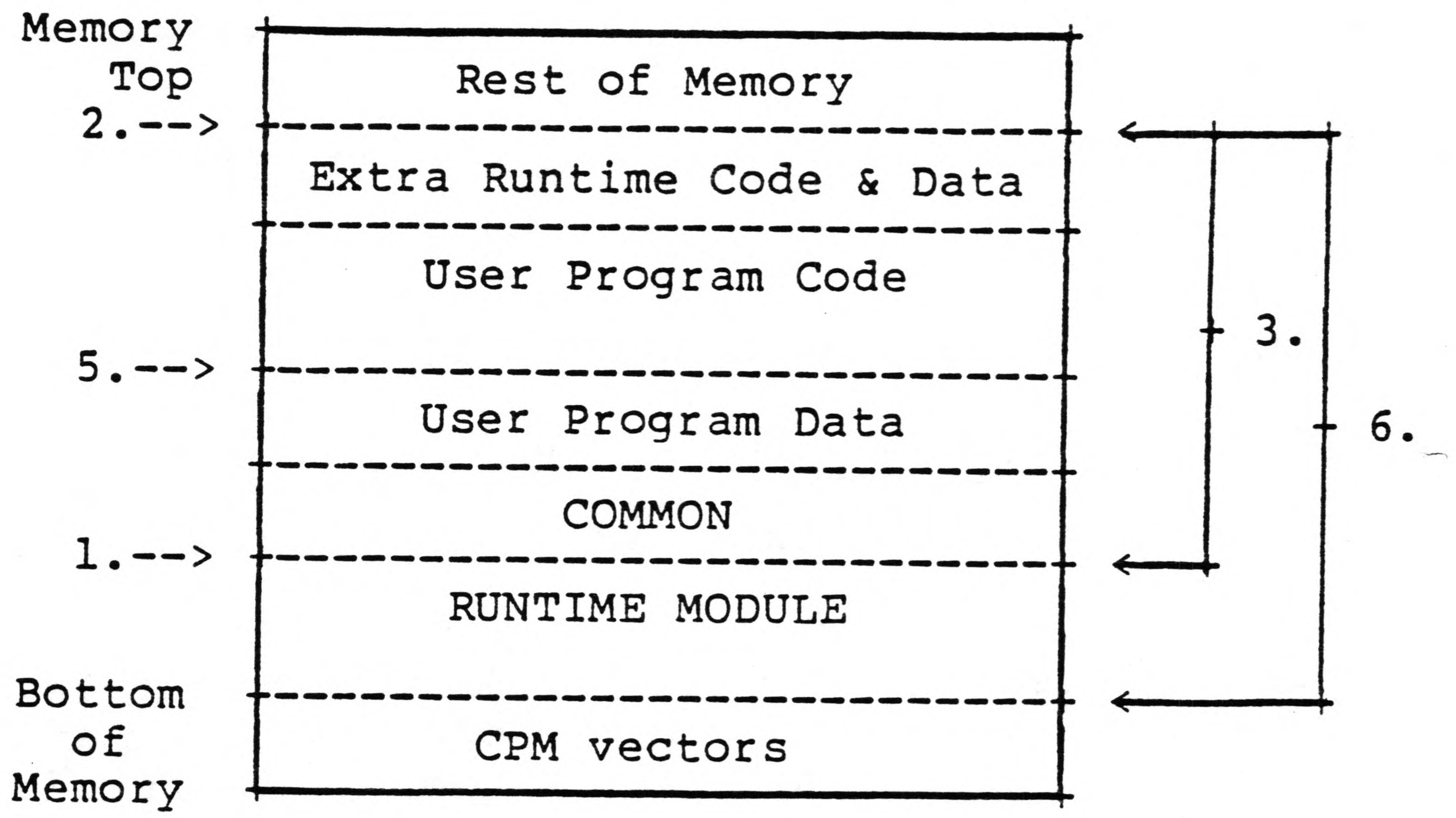


Figure 7.1 Link Data Map

1. <program-start> - Hexadecimal address of the beginning of your program.

2. <program-end> - Hexadecimal address of the end of your program.

3. <bytes> - Decimal size of program in bytes.

4. <free-bytes> - Decimal size of unused memory in bytes during linking.

5. <start-address> - Hexadecimal start address of your program (not necessarily the same as <program-start>).

6. <num-of-pages> - Decimal number of 256-byte pages used by program.

For programs linked to BASLIB.REL and using the BRUN.COM runtime module, the size of your .COM file in bytes is equal to:

```
<program-end> - <start-address> + 128
```

At runtime, remember that BRUN.COM also resides in memory along with your program. The 128 bytes in the above equation is for a small relocator routine that begins every .COM file. When you invoke a program, this relocator routine is the first routine executed. All it does is move the rest of your .COM file to the start address shown above. Execution of your program then begins. The first thing your program does is load the runtime module to establish the runtime support environment.

We now discuss linking to compiled BASIC .REL files.

7.2 LINKING TO COMPILED BASIC .REL FILES

Because of the way the BASIC runtime environment is implemented with the BRUN.COM runtime module, there are a number of peculiarities that you must account for at linktime.

First of all, before you can link any BASIC .REL file, you must have the file BCLOAD on the currently logged disk. BCLOAD contains two pieces of information: the hexadecimal load address of your program, and the drive in which to find BRUN.COM at runtime.

BCLOAD looks like this if you TYPE it out:

```
+4000     [Program Load Address]
:         [A:, B:, C:, etc., or : for default]
```

At runtime, you must have BRUN.COM on the disk specified in BCLOAD or an error is generated. Note that the plus sign (+) is necessary to tell the linker to write the .COM file beginning at the start address of your program instead of the program load address. (The start address is the address at which your program begins execution.) The default location of the BRUN.COM runtime module is the currently logged drive. You can alter BCLOAD, before <u>linktime</u>, to specify the disk on which you want BRUN.COM to reside at <u>runtime</u>.

There are two other peculiarities associated with linking programs that require the BRUN.COM runtime module. Namely, these linking procedures may not work:

```
L80 FOO/G

L80 FOO/E followed by SAVE xxx FOO.COM
```

L80 FOO/G may not work if BRUN.COM does not reside on the disk you have specified in BCLOAD. CHAINing of programs does not work properly if you use SAVE after a link.

We now move to a discussion of the runtime support that is linked to your program.

7.3 RUNTIME SUPPORT

Once you have compiled a .REL file, you need to link your program to modules that contain runtime support routines. Runtime support is the body of routines that, in essence, implement the BASIC language. Your compiled .REL file, on the other hand, implements the particular algorithm that makes your program a unique BASIC program.

Runtime support is essential to the execution of all compiled BASIC programs. It is found in BRUN.COM and the runtime library. As a rule, only a portion of all possible runtime routines is linked to your .REL file. The length of time necessary to link all these needed runtime support routines is often a problem on microcomputers.

Partly for this reason, the BRUN.COM runtime module contains all of the more frequently used routines in one module. Since they all reside in one module, they are linked all at once, and need not be searched for in later linker searches. Note that the BRUN.COM module is automatically linked to every program via a dummy module in BASLIB.REL: it is not present in memory at linktime. Thus, a minimal program at runtime is at least 16K long. If your program uses other less frequently used routines, these routines are searched for and found in BASLIB.REL. At linktime, you cannot use the /P and /D linker switches, since they will cause errors at program runtime.

When you specify the /O switch at <u>runtime</u>, the alternate runtime library (OBSLIB.REL) is substituted for BASLIB.REL as the default library to be searched at <u>linktime</u>. At linktime you can then use /P and /D as described in the Utility Software Manual. Note that when OBSLIB.REL is selected as the library to be searched, BRUN.COM is not used by your program at all.

There are several advantages to using OBSLIB.REL:

1. Programs not using BRUN.COM can be put in ROM, since separate instruction and data areas can be created when linking to routines in OBSLIB.REL with the /P and /D switches.

2. For small and simple programs, you may be able to compile and link smaller programs than the 16K minimum required to accommodate the BRUN.COM module. This can be of importance in compiling a program for a ROM-based application, where space can be a critical factor.

3. Execution of a compiled and linked .COM file does not require the existence of BRUN.COM on disk at runtime.

There are, however, some distinct disadvantages to using OBSLIB.REL:

1. COMMON is not supported between programs.

2. The CHAIN command is semantically equivalent to the RUN command.

3. COMMON and CHAIN commands cannot be used to support a system of programs sharing common data. (See 1. and 2. above.)

4. The CLEAR command is not implemented.

5. The RUN <linenumber> option to RUN is not implemented.

6. The linker cannot load programs as large as those that use the BRUN.COM module.

7. All required runtime support functions are included in every .COM file generated, thus increasing the size of each of your .COM files. This is not the case for .COM files using the BRUN.COM runtime module.

For more information on using CHAIN and COMMON with a system of programs, see Appendix A. For more information on ROM-able code, see Appendix B.

CHAPTER 8

RUNNING A COMPILED PROGRAM

To run a compiled program, simply enter the filename without its .COM filename extension. For example:

```
DEMO
```

The above command causes execution of the program DEMO.COM. At runtime, BRUN.COM must be accessible from disk. BRUN.COM is loaded from the disk in the drive you specify in BCLOAD at linktime.

Programs can also be executed immediately after linking is complete by using the /G linker switch. This works only if BRUN.COM is on the disk you have selected in BCLOAD.

The executable binary file can also be executed from within a program, as in the following statement:

```
10 RUN "PROG"
```

The default extension is .COM. The .COM file can be a program created in any programming language. The CHAIN command is used in a similar fashion. In either case, an executable binary file is loaded. The BRUN.COM runtime module is not reloaded when you use CHAIN; it is when you use RUN.

It is important to realize that the bulk of the runtime environment is taken up by the BRUN.COM runtime module. This module is automatically loaded when you initially invoke an executable .COM file requiring BRUN.COM. When you RUN a program, the .COM file is loaded into memory and BRUN.COM is also loaded to create a fresh runtime environment. Both files reside in memory simultaneously.

CHAPTER 9

A COMPILER/INTERPRETER COMPARISON

There are differences between the languages supported by the BASIC Compiler and the BASIC-80 interpreter that must be taken into account when compiling existing or new BASIC programs. This is why we strongly recommend that you compile the demonstration program in Chapter 3 first; read Chapters 4-8; and only then begin compiling other programs.

The differences between the languages supported by the BASIC Compiler and the BASIC interpreter fall into three categories: operational differences, language differences, and other differences. The tables on the next page serve as a reference guide to these differences. All commands and functions except %INCLUDE are described in the BASIC-80 Reference Manual. Where differences exist, those commands and functions are also discussed in the following paragraphs.

9.1 OPERATIONAL DIFFERENCES

Those BASIC-80 commands used to operate in the BASIC-80 programming environment are not acceptable input to the compiler. These include the following:

AUTO	CLOAD	CSAVE	CONT	DELETE
EDIT	LIST	LLIST	LOAD	MERGE
NEW	RENUM	SAVE		

9.2 LANGUAGE DIFFERENCES

Most programs that run under the BASIC-80 interpreter will compile under the BASIC Compiler with little or no change. However, it is necessary to note differences in the following commands:

CALL	%INCLUDE
CHAIN	ON ERROR GOTO
CLEAR	REM
COMMON	RESUME
DEFxxx	RUN
DIM	STOP
END	TRON/TROFF
ERASE	USRn
FOR/NEXT	WHILE/WEND

These differences are described below:

1. <u>CALL</u>
   The CALL statement allows you to call and transfer program control to a precompiled FORTRAN-80 subroutine, or to an assembly language routine that you have created with MACRO-80. The format of the CALL Statement is:

   ```
   CALL <variable-name> [<argument-list>...]
   ```

   The <variable-name> parameter is the name of the subroutine that you wish to call. This name must be 1 to 6 characters long and must be recognized by LINK-80 as a global symbol. That is, <variable-name> must be the name of the subroutine in a FORTRAN SUBROUTINE statement, or a PUBLIC symbol in an assembly language routine. Refer to the MACRO-80 Reference Manual and the FORTRAN-80 Reference Manual for definitions of these terms. (See NOTE below.)

The <argument-list> parameter is optional. It contains arguments that are passed to an assembly language or FORTRAN subroutine.

Example: 120 CALL MYSUBR (I,J,K)

NOTE

FORTRAN-80 is a separate product available from Microsoft and is not part of the BASIC Compiler package. If you do not have FORTRAN-80, then the CALL statement can only be used with assembly language subroutines.

Further information on assembly language subroutines is contained in in the discussion of the USR function that follows in this chapter. Also, more information is provided on creating and interfacing assembly language routines in the Utility Software Manual.

2. CHAIN
   The BASIC Compiler does not support the ALL, MERGE, DELETE, and <line number> options to CHAIN. If you wish to pass variables, it is recommended that the COMMON statement be used. Note that files are left open during CHAINing.

3. <u>CLEAR</u>
The BASIC Compiler supports the CLEAR command as described in the BASIC-80 Reference Manual, with the restriction that <expression1> and <expression2> must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used. The default stack size is 256 bytes and the default top of memory is the current top of memory. The CLEAR statement performs the following actions:

    Closes all files
    Clears all COMMON and user variables
    Resets the stack and string space
    Releases all disk buffers

See Appendix C for a memory map showing the location of the stack, string space, and disk buffers discussed above.

Note that CLEAR is supported only for programs using the BRUN.COM module, and not for programs linked to the OBSLIB.REL runtime library.

4. <u>COMMON</u>
The BASIC Compiler supports a modified version of the COMMON statement. The COMMON statement must appear in a program before any executable statements. A list of <u>non</u>-executable statements follows:

    COMMON
    DEFDBL, DEFINT, DEFSNG, DEFSTR
    DIM
    OPTION BASE
    REM
    %INCLUDE

All other statements are executable. Arrays in COMMON must be declared in preceding DIM statements.

The standard form of the COMMON statement is referred to as blank COMMON. FORTRAN-style named COMMON areas are also supported; however, the named COMMON variables are not preserved across CHAINs.

The syntax for named COMMON is as follows:

    COMMON /<name>/ <list of variables>

The parameter <name> is 1 to 6 alphanumeric characters starting with a letter. This is useful for communicating with FORTRAN and assembly

language routines without having to explicitly pass parameters in the CALL statement.

IMPORTANT

> For blank COMMON statements communicating between CHAINing and CHAINed-to programs, both the size of the COMMON area, and the order of variables must be the same.

To ensure that COMMON areas can be shared between programs, place blank COMMON declarations in a single INCLUDE file and use the %INCLUDE statement in each program. For example:

```
MENU.BAS
10 %INCLUDE  COMDEF
      .
      .
      .
1000 CHAIN "PROG1"


PROG1.BAS
10 %INCLUDE  COMDEF
      .
      .
      .
2000 CHAIN "MENU"


COMDEF.BAS
100 DIM A(100),B$(200)
110 COMMON I,J,K,A()
120 COMMON A$,B$(),X,Y,Z
```

5. DEFINT/SNG/DBL/STR
DEFxxx statements designate the storage class and data type of variables listed as parameters. The compiler does not "execute" DEFxxx statements as it does a PRINT statement, for example.

Instead, the compiler allocates memory for storage of designated variables, and assigns them one of the following data types:

1. INTeger,

2. SiNGle precision floating point,

3. DouBLe precision floating point, or

4. STRing.

A DEFxxx statement takes effect as soon as it is encountered in your program during compilation. Once the type has been defined for the listed variables, that type remains in effect either until the end of the program or until another DEFxxx statement alters the type of the variable. Unlike the interpreter, the compiler cannot circumvent the DEFxxx statement by directing flow of control around it with a GOTO. For variables given with a precision designator (i.e., %, !, #, as in A%=B), the type is not affected by the DEFxxx statement.

6. DIM
The DIM statement is similar to the DEFxxx statement in that it is scanned rather than executed. That is, DIM takes effect when it is encountered at compiletime and remains in effect until the end of the program: it cannot be re-executed at runtime. If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, an "Array Already Dimensioned" error results. Therefore, the practice of putting a collection of DIM statements in a subroutine at the end of your program generates fatal errors. In that case, the compiler sees the DIM statement only after it has already assigned the default dimension to arrays declared earlier in the program.

Also note that the values of the subscripts in a DIM statement must be integer constants; they may not be variables, arithmetic expressions, or floating point values. For example, each of the following DIM statements is illegal:

```
DIM A1(I)
DIM A1(3+4)
DIM A1(3.4E5)
```

7. <u>END</u>
During execution of a compiled program, an END statement closes files and returns control to the operating system. The compiler assumes an END statement at the end of the program, so "running off the end" (omitting an END statement at the end of the program) produces proper program termination by default.

8. <u>ERASE</u>
The ERASE statement is not implemented for the compiler. ERASE in BASIC-80 allows you to re-dimension arrays, something that is not done in the compiled environment.

9. <u>FOR/NEXT</u>
Double precision FOR/NEXT loops can be used with the compiler. Also, FOR/NEXT loops must be statically nested. Static nesting means that each FOR must have a single corresponding NEXT.

Static nesting also means that each FOR/NEXT pair must reside within an outer FOR/NEXT pair. Therefore, the following construction is not allowed:

```
FOR I
    FOR J
        FOR K

    NEXT J
        NEXT K
NEXT I
```

This construction is the correct form:

```
FOR I
    FOR J
        FOR K

        NEXT K
    NEXT J
NEXT I
```

Also, you should not direct program flow into a FOR/NEXT loop with a GOTO statement. The result of such a jump is undefined, as in the following example:

```
50  GOTO 100
        :
        :
90  FOR I = 1 to 10
        :
100 PRINT "INLOOP"
        :
200 NEXT I
```

10. <u>%INCLUDE</u>

The format of the %INCLUDE compiler directive is:

%INCLUDE <filename>

%INCLUDE allows the compiler to include source code from an alternate BASIC file. These BASIC source files may be subroutines, single lines, or any type of partial program. No assembly language or FORTRAN files are allowed as arguments to the %INCLUDE statement. Note that <filename> does <u>not</u> require quotes and that the default extension <u>is</u> .BAS.

The programmer should take care that any variables in the included files match their counterparts in the main program, and that included lines do not contain GOTOs to non-existent lines, END statements, or similarly erroneous code.

These further restrictions must be observed:

(a.) The INCLUDEd file must be SAVEd with the ,A option if created from within BASIC-80.

(b.) The INCLUDEd lines must be in ascending order.

(c.) The lowest line number of the included lines must be higher than the line number of the INCLUDE statement in the main program.

(d.) The range of line numbers in the INCLUDEd file must numerically precede subsequent line numbers in the main program. These restrictions are removed if the main program is compiled with the /C switch set, since line numbers need not be in ascending order in this case. For more information, see Section 6.3, Compiler Switches.

(e.) %INCLUDE directives cannot be nested inside INCLUDE files. This means that %INCLUDE can only be used in the file containing your main BASIC program.

(f.) The %INCLUDE directive must be the last statement on a line, as in the following statement:

```
999  DEFINT I-N : %INCLUDE COMMON.BAS
```

11. <u>ON ERROR GOTO</u>
    If a program contains ON ERROR GOTO and RESUME <line number> statements, the /E compilation switch must be given in the compiler command line. If the RESUME NEXT, RESUME, or RESUME 0 form is used, the /X switch must be used instead.

    The basic function of these switches is to allow the compiler to function correctly when error trapping routines are included in a program. See Section 6.3, Compiler Switches, for a detailed explanation of these switches. Note, however, that the use of these switches increases the size of the .REL and .COM files.

12. <u>REM</u>
    REM statements are REMarks starting with a single quotation mark or the word REM. Since REM statements do not take up time or space during execution, REM may be used as freely as desired. This practice is encouraged for improving the readability of your programs.

13. <u>RESUME</u>
    See the preceding discussion of ON ERROR GOTO.

14. <u>RUN</u>
    The compiler supports both the RUN and RUN <line number> forms of the RUN statement. The BASIC Compiler does not support the "R" option with RUN. If this feature is desired, the CHAIN statement should be used. Note that RUN is used to execute .COM files created by the BASIC Compiler, and does not support the execution of BASIC source files as does the interpreter.

    Other .COM files not created with the BASIC Compiler <u>are</u> executable with the RUN statement. These can be .COM files created in other languages besides BASIC.

15. STOP
The STOP statement is identical to the END statement, except that it terminates your program at a point that is not necessarily its end. It also prints a message telling you at which hexadecimal address you have stopped. If the /D, /E, or /X compiler switches are turned on, then the message prints the line number at which you have stopped. As with the END statement, STOP closes all open files and returns control to the operating system. STOP is normally used for debugging purposes.

16. TRON/TROFF
In order to use TRON/TROFF, the compiler /D Debug switch must be switched on. Otherwise, TRON and TROFF are ignored and a warning message is generated.

17. USRn Functions
Although the USRn function is implemented in the compiler to call machine language subroutines, there is no way to pass parameters, except through the use of POKEs to protected memory locations that are later accessed by the machine language routine.

When the compiler sees X = USRn (0), it generates the following code:

```
CALL    $U%+const
SHLD    X%
```

If you have compiled the program with the /Z switch on, then the compiler generates instead similar Z80 code:

```
CALL    $U%+const
LD      (X%),HL
```

During execution, the program encounters this code, jumps to the address of the CALL, performs the steps of your subroutine and returns. Your routine should place the integer result of the routine in the H,L register pair prior to returning to the compiled BASIC program. On return, as shown above, the contents of the H,L register pair are placed in the location of the variable X. Any other parameters to be passed must be PEEKed from the main BASIC program, and POKEd into protected memory locations. With this method of passing parameters, the USRn function is quite usable. You must take responsibility, though, to ensure that your code and any variables you use are protected.

If you do not want to use the above method of passing parameters, you have two other choices:

1. If your machine language routine is short enough, you can store it by making the <u>first</u> string defined in the program contain the ASCII values corresponding to the hexadecimal values of your routine. Use the CHR$ function to insert ASCII values in the string. You can then find the start of your routine by using the VARPTR function. For example, for the string A$, VARPTR (A$) will return the address of the length of the string. The next two addresses are (first) the least significant byte and (then) the most significant byte of the actual address of the string. This set-up of the string space for the compiler differs from the set-up for the interpreter in this respect. Thus, to find the actual start address of your routine, you would use the following BASIC instructions:

```
A$ = "String containing routine"
I% = VARPTR(A$)
AD% = PEEK(I% + 2) * 256 + PEEK(I% + 1)
AD%  is the start address of your routine.
```

   Note that strings move around in the string space, so any absolute references must be adjusted to reflect the current memory location of the routine. To make your code position independent for the Z80, you should use relative, rather than absolute jumps.

2. The second method is to reset the default value of the load address in the BCLOAD file. The BCLOAD file's main purpose is to direct loading of your executable program in memory after BRUN.COM has been loaded. By increasing the load address by 100H, for example, 256 bytes of free protected space are created between the end of BRUN.COM and the start of the loading area. Machine language routines or data can then be safely POKEd into this area.

   A better alternative is to use MACRO-80 to assemble your subroutines. Then, your subroutines can be linked directly to the compiled program and referenced using the CALL statement.

18. WHILE/WEND
WHILE/WEND constructions should be statically nested. Static nesting means that each WHILE/WEND pair, when nested within other FOR/NEXT or WHILE/WEND pairs, cannot reside partly in, and partly outside, the nesting pair. For example, the following construction is not allowed:

```
FOR I = 1 to 10
   A = COUNT
   WHILE A = 1
NEXT I
   A = A - 1
   WEND
```

You should also not direct program flow into a WHILE/WEND loop without entering through the WHILE statement. See FOR/NEXT, above, for an example of this restriction.

9.3 OTHER DIFFERENCES

Other differences between BASIC-80 and the BASIC Compiler include the following:

1. Expression Evaluation - The BASIC Compiler performs optimizations, if possible, when evaluating expressions.

2. Use of Integer Variables - The BASIC Compiler can make optimum use of integer variables as loop control variables. This allows some functions (and programs) to execute up to 30 times faster than when interpreted.

3. Double Precision Arithmetic Functions - The BASIC Compiler implements double precision arithmetic functions, including all of the transcendental functions.

4. String Space Implementation - To increase the speed of garbage collection, the implementation of the string space for the compiler differs from its implementation for the interpreter.

EXPRESSION EVALUATION

During expression evaluation, the BASIC Compiler converts operands of different types to the type of the more precise operand.

```
QR=J%+A!+Q#
```

The above expression causes J% to be converted to single precision and added to A!. This double precision result is added to Q#.

The BASIC Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 10000 for M%.

```
I%=20000
J%=20000
K%=-30000
M%=I%+J%-K%
```

That is, J% is added to I%. Because the number is too large, it converts the result into a floating point number. K% is then coverted to a floating point number and subtracted. The result, 10000, is found, and converted back to an integer and saved as M%.

The BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow occurs. If the /D Debug switch is set, the error is detected. Otherwise, an incorrect answer is produced.

Besides the above type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

```
I%=20000
J%=-18000
K%=20000
M%=I%+J%+K%
```

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence, and parentheses may be used to direct the order of evaluation. <u>No other guarantee of evaluation order can be made</u>.

INTEGER VARIABLES

To produce the fastest and most compact object code possible, you should make maximum use of integer variables. For example, the following program executes approximately 30 times faster by replacing "I", the loop control variable, with "I%" or by declaring I an integer variable with DEFINT.

```
FOR I=1 TO 10
A(I)=0
NEXT I
```

Also, it is especially advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

DOUBLE PRECISION ARITHMETIC FUNCTIONS

The BASIC Compiler allows you to use double precision floating point numbers as operands for arithmetic functions, including all of the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, SQR). Only single precision arithmetic functions are supported by the interpreter.

Your program development strategy when designing a program with double precision arithmetic functions should be the following:

1. Implement your BASIC program using single precision operands for all functions that you later intend to be double precision.

2. Debug your program with the interpreter to determine the soundness of your algorithm before converting variables to double precision.

3. Declare all desired variables as double precision. Your algorithm should be sound at this point.

4. Compile and link your program. It should implement the algorithm that you have already debugged with the interpreter, now with double the precision in your arithmetic functions.

STRING SPACE IMPLEMENTATION

The compiler and interpreter differ in their implementation and maintenance of the string space. Using PEEK, POKE, VARPTR, or assembly language routines to change string descriptors may result in a String Space Corrupt error. See more information on the string space in the discussion of the USR function earlier in this chapter.

CHAPTER 10

ERROR MESSAGES

During development of a BASIC program with the BASIC Compiler, three different kinds of errors may occur: BASIC Compiler fatal errors, BASIC Compiler warning errors, and BASIC runtime errors. This chapter lists error codes, error numbers, and error messages for each type of error.

10.1 BASIC COMPILETIME ERROR MESSAGES

For errors that occur at compiletime, the compiler outputs the line containing the error, an arrow beneath that line pointing to the place in the line where the error occurred, and a two-character code for the error. In some cases, the compiler reads ahead on a line to determine whether an error has actually occurred. In those cases, the arrow points a few characters beyond the error, or to the end of the line.

The BASIC Compiletime errors described below are divided into Fatal Errors and Warning Errors.

FATAL ERRORS

CODE	MESSAGE
BS	Bad Subscript Illegal dimension value Wrong number of subscripts
CD	Duplicate COMMON variable
CN	COMMON array not dimensioned
CO	COMMON out of order
DD	Array Already Dimensioned
FD	Function Already Defined
FN	FOR/NEXT Error FOR loop index variable already in use FOR without NEXT NEXT without FOR
IN	INCLUDE Error %INCLUDE file not found
LL	Line Too Long
LS	String Constant Too Long
OM	Out of Memory Array too big Data memory overflow Too many statement numbers Program memory overflow
OV	Math Overflow
SN	Syntax error - caused by one of the following: Illegal argument name Illegal assignment target Illegal constant format Illegal debug request Illegal DEFxxx character specification Illegal expression syntax Illegal function argument list Illegal function name Illegal function formal parameter Illegal separator Illegal format for statement number Illegal subroutine syntax Invalid character Missing AS Missing equal sign

Missing GOTO or GOSUB
Missing comma
Missing INPUT
Missing line number
Missing left parenthesis
Missing minus sign
Missing operand in expression
Missing right parenthesis
Missing semicolon
Missing slash
Name too long
Expected GOTO or GOSUB
String assignment required
String expression required
String variable required
Illegal syntax
Variable required
Wrong number of arguments
Formal parameters must be unique
Single variable only allowed
Missing TO
Illegal FOR loop index variable
Illegal COMMON name
Missing THEN
Missing BASE
Illegal subroutine name

SQ Sequence Error
Duplicate statement number
Statement out of sequence

TC Too Complex
Expression too complex
Too many arguments in function call
Too many dimensions
Too many variables for LINE INPUT
Too many variables for INPUT

TM Type Mismatch
Data type conflict
Variable must be of same type

UC Unrecognizable Command
Statement unrecognizable
Command not implemented

UF Function Not Defined

WE WHILE/WEND Error
WHILE without WEND
WEND without WHILE

/0 Division by Zero

/E	Missing "/E" Switch
/X	Missing "/X" Switch

WARNING ERRORS

CODE	MESSAGE
ND	Array not Dimensioned
SI	Statement Ignored Statement ignored Unimplemented command

10.2 BASIC RUNTIME ERROR MESSAGES

The following errors may occur at program runtime. The error numbers match those issued by the BASIC-80 interpreter. The compiler runtime system prints long error messages followed by an address, unless /D, /E, or /X is specified in the compiler command line. In those cases, the error message is also followed by the number of the line in which the error occurred.

NUMBER	MESSAGE
2	Syntax Error A line is encountered that contains an incorrect sequence of characters in a DATA statement.
3	RETURN without GOSUB A RETURN statement is encountered for which there is no previous, unmatched GOSUB statement.
4	Out of Data A READ statement is executed when there are no DATA statements with unread data remaining in the program.
5	Illegal Function Call A parameter that is out of range is passed to a math or string function. A function call error may also occur as the result of: A negative or unreasonably large subscript A negative or zero argument with LOG A negative argument to SQR A negative mantissa with a non-integer exponent A call to a USR function for which the starting address has not yet been given An improper argument to ASC, CHR$, MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING$, SPACE$, INSTR, or ON...GOTO A string concatenation that is longer than 255 characters
6	Floating Overflow or Integer Overflow The result of a calculation is too large to be represented within the range allowed for floating point numbers.

9 Subscript Out of Range
An array element is referenced with a subscript that is outside the dimensions of the array.

11 Division by Zero
A division by zero is encountered in an expression, or the operation of involution results in zero being raised to a negative power.

14 Out of String Space
String variables exceed the allocated amount of string space.

20 RESUME without Error
A RESUME statement is encountered before an error trapping routine is entered.

21 Unprintable Error
An error message is not available for the error condition that exists. This is usually caused by an ERROR with an undefined error code.

50 Field Overflow
A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file.

51 Internal Error
An internal malfunction occurs in the BASIC Compiler. Report to Microsoft the conditions under which the message appeared.

52 Bad File Number
A statement or command references a file with a file number that is not OPEN or is out of the range of file numbers specified at initialization.

53 File Not Found
A LOAD, KILL, or OPEN statement references a file that does not exist on the current disk.

54 Bad File Mode
An attempt is made to use PUT, GET, or LOF with a sequential file, to LOAD a random file, or to execute an OPEN with a file mode other than I, O, or R.

55 File Already Open
A sequential output mode OPEN is issued for a file that is already open; or a KILL is given for a file that is open.

57 Disk I/O Error
An I/O error occurred on a disk I/O operation. The operating system cannot recover from the error.

58 File Already Exists
The filename specified in a NAME statement is identical to a filename already in use on the disk.

61 Disk Full
All disk storage space is in use.

62 Input Past End
An INPUT statement reads from a null (empty) file, or from a file in which all data has already been read. To avoid this error, use the EOF function to detect the end of file.

63 Bad Record Number
In a PUT or GET statement, the record number is either greater than the maximum allowed (32767) or is equal to 0.

64 Bad File Name
An illegal form is used for the filename with LOAD, SAVE, KILL, or OPEN (e.g., a filename with too many characters).

67 Too Many Files
The 255 file directory maximum is exceeded by an attempt to create a new file with SAVE or OPEN.

The following additional runtime error messages are fatal and cannot be trapped:

Internal Error - String Space Corrupt

Internal Error - String Space Corrupt during G.C.

Internal Error - No Line Number

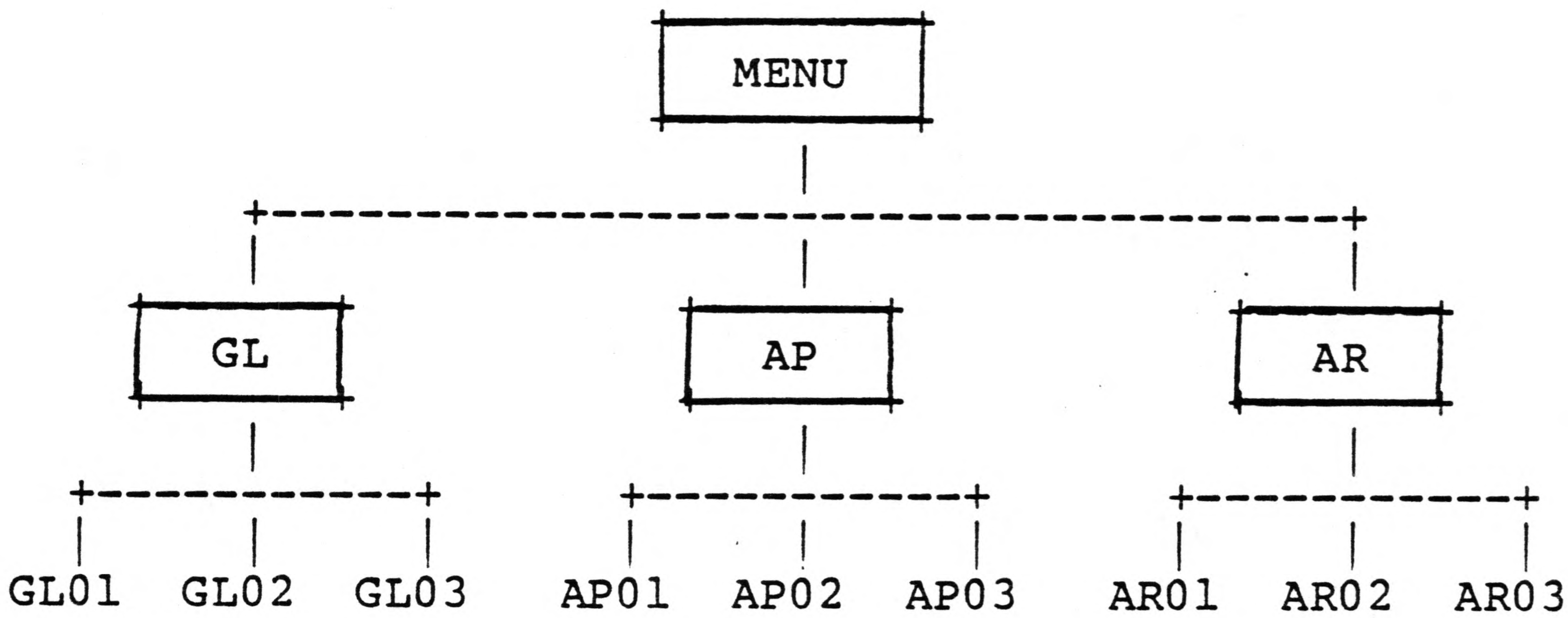
The first two errors usually occur because a string descriptor has been improperly modified. (G.C stands for garbage collection.) The last error occurs when the error address cannot be found in the line number table during error trapping.

APPENDIX A

Creating a System of Programs with the BRUN.COM Runtime Module

The CHAINing with COMMON feature and the BRUN.COM runtime module are designed for creating large systems of BASIC programs that interact with each other. A hypothetical system will be described to show the interactions in a large system design. In particular, the distinction between CHAIN and RUN will be highlighted.

Consider the following integrated accounting system containing three packages for general ledger, accounts payable, and accounts receivable. Entry into each package is controlled by a main menu program. The system structure is shown below:



In order to use CHAINing with COMMON effectively, it is important to logically structure the system and the COMMON information. In the system pictured above, COMMON information exists within each of the packages GL, AP, and AR. Each package contains a system of three separately compiled programs. Furthermore, there may be COMMON information between MENU and each of the packages. Note that there may be overlapping sets of COMMON information. The compiler's COMMON statement is not as flexible as the interpreter's: COMMON areas must be the same size in programs that CHAIN to each other.

Two solutions to this problem of communicating between programs are given below, though others are possible:

1. Use the same COMMON declarations in all programs so that all common information may be shared, or

2. Use the same set of COMMON declarations within each of the three packages with no common information shared via COMMON with the other packages or the main MENU program. In this case, there are three sets of COMMON declarations, one for each package.

For a large integrated set of systems of programs, the second method gives more flexibility with the compiler. Since program control is switched from package to package through the main MENU, there is little loss of flexibility with this method. Any common information that could be obtained in MENU should be obtained in the main program for each of the packages GL, AP, and AR. This is the same approach you would use with a single package.

For the above diagram, the use of CHAIN and RUN in each of the major programs is outlined in the following program fragments:

```
MENU.BAS
        1000 If MENU=1 THEN RUN "GL"
        1010 IF MENU=2 THEN RUN "AP"
        1020 IF MENU=3 THEN RUN "AR"

GL.BAS                               General Ledger
        10 %INCLUDE GLCOMDEF         (GL) COMMON declarations
        1000 CHAIN "GL01"
        1010 CHAIN "GL02"
        1020 CHAIN "GL03"
        1030 IF MENU=YES THEN RUN "MENU"

AP.BAS                               Accounts Payable
        10 %INCLUDE APCOMDEF         (AP) COMMON declarations
        1000 CHAIN "AP01"
        1010 CHAIN "AP02"
        1020 CHAIN "AP03"
        1030 IF MENU=YES THEN RUN "MENU"

AR.BAS                               Accounts Receivable
        10 %INCLUDE ARCOMDEF         (AR) COMMON declarations
        1000 CHAIN "AR01"
        1010 CHAIN "AR02"
        1020 CHAIN "AR03"
        1030 IF MENU=YES THEN RUN "MENU"
```

Each of the lower level programs XXYY (XX=GL, AP, AR, YY = 01, 02, 03) should CHAIN back to the package main program XX.

The RUN statement in the above programs loads the specified program as a normal .COM file and starts execution. For compiled BASIC programs, a new copy of the BRUN.COM runtime module is reloaded. This allows a new system of CHAINed programs to be started. During CHAINs, the BRUN.COM runtime module is in control, like the BASIC interpreter during interpretation, and BRUN.COM is not reloaded.

APPENDIX B

ROM-able Code

To create a program that can be burned into ROM, you should note the following:

1. Constant data and instructions can go into ROM.

2. Variable data cannot go into ROM.

Therefore, it is necessary that ROM-able code have separate data and instruction areas. You can specify these areas at linktime by using the /D and /P switches (D for Data and P for Program). See the Utility Software Manual for more information on the use of these switches.

Unfortunately, you cannot use the /P and /D switches if you choose to link a program that uses the BRUN.COM runtime module. Furthermore, any program that requires BRUN.COM cannot be put into ROM.

The only way that you can put a compiled BASIC program into ROM is by linking to the OBSLIB.REL runtime library. This library is searched by default at <u>linktime</u> only if at <u>compiletime</u> you compile with the /O switch.

The disadvantages of using OBSLIB.REL are discussed in Chapter 7.

APPENDIX C

Memory Map

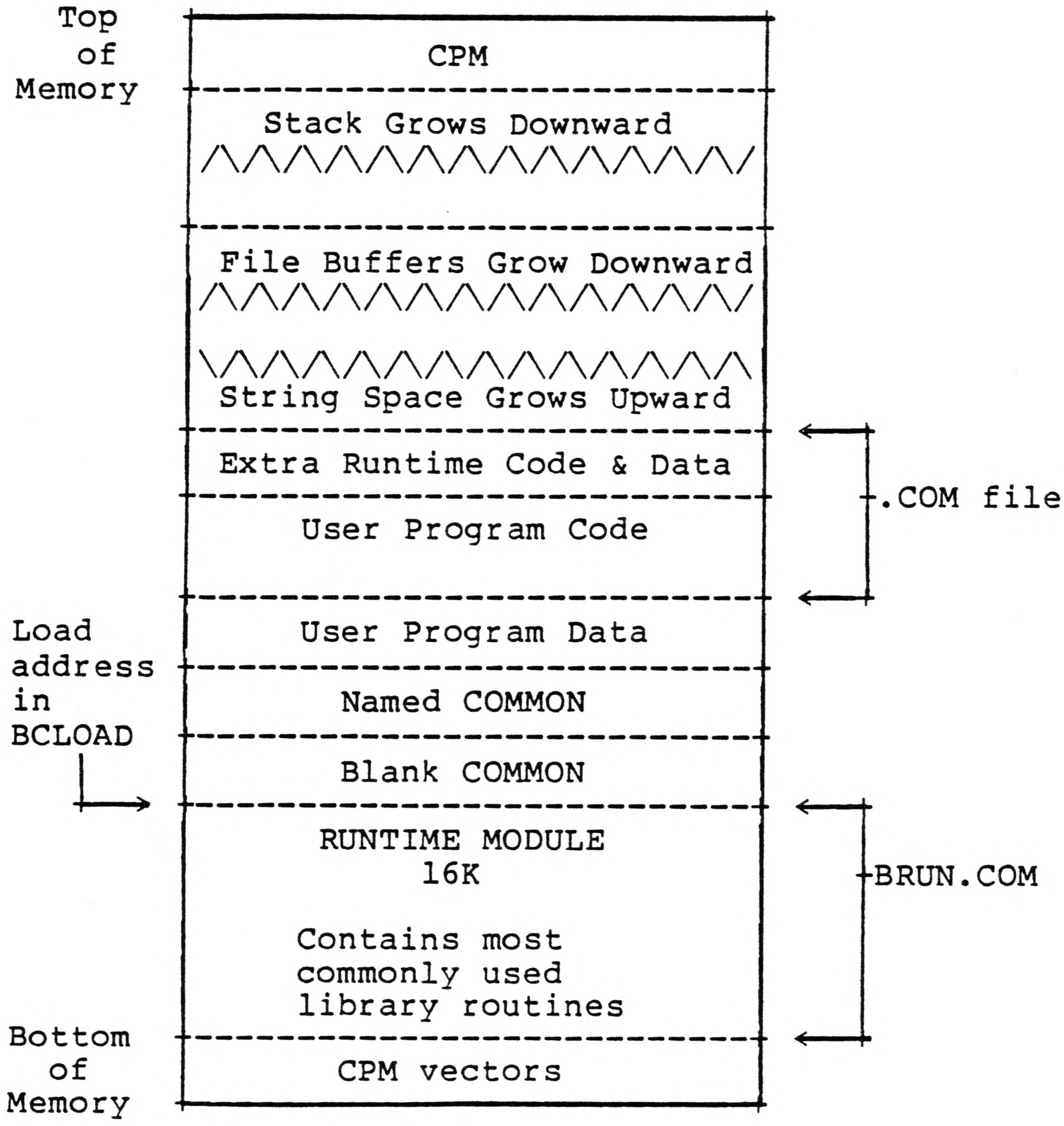


Figure 1 Runtime Memory Map

Runtime memory map of a program using the BRUN.COM runtime module.

APPENDIX D

Differences between Version 5.3 and Previous Versions of the BASIC Compiler

Described below are the major differences between this version of the BASIC Compiler (5.3) and previous versions of the compiler:

1. Your compiled programs now rely on a large runtime module for most of the runtime support that you need during program execution. This module is named BRUN.COM.

2. What used to be called BASLIB.REL, is now called OBSLIB.REL (short for Old BaSlib). The runtime library on your disk called BASLIB.REL contains a dummy module containing references to all the routines in the BRUN.COM module. BRUN.COM is never in memory at linktime.

3. The COMMON statement now works between CHAINed subprograms, as well as between functions in the same program.

4. The CHAIN statement is no longer semantically equivalent to RUN, and true chaining is allowed. Note that CHAIN <filename> does not cause reloading of the runtime module. In fact, BRUN.COM acts much like the interpreter in this case, supervising the change of control from one program to the next.

5. The CLEAR command is now implemented.

6. The RUN <line-number> form of the RUN command is now implemented.

As a result of the above changes to the BASIC compiler package, the royalty requirements have been altered. The old runtime library (what used to be BASLIB.REL and is now OBSLIB.REL) can be used in your applications without payment of royalties. However, notice must exist within your application that portions of your software are copyrighted by Microsoft.

However, any distribution of the BRUN.COM runtime module requires payment of royalties. Examine your non-disclosure agreement or contact Microsoft for more specific information on the nature of royalty payments.

INDEX